AFIT/GCS/ENG/GCS93D-18

A Prototype Architecture for an Automated Scenario Generation System for Combat Simulations

THESIS
Mark W. Pfefferman
Captain, USAR

AFIT/GCS/ENG/GCS93D-18

AFIT/GCS/ENG/GCS93D-18

A Prototype Architecture for an Automated Scenario Generation System for Combat Simulations

THESIS

Presented to the Faculty of the Graduate School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science

Mark W. Pfefferman, B.S.
Captain, USAR

December, 1993

Approved for public release; distribution unlimited

Preface

First of all, I would like to thank my thesis committee - Major Dave Sonnier, Dr. Thomas Hartrum, Major Gregg Gunsch and Major Ed Negrelli - for all of their excellent advice (and sometimes brutally honest comments). I really appreciate the chance to complete my thesis after leaving active duty.

I also want to thank Captain Seth Guanu for all his help in testing the BATTLESIM scenario files that I generated.

Last, but certainly not least, I would like to thank my family. My wife Cindy has put up with so much over the years from me and work. Life here at AFIT was extremely challenging for me and Cindy somehow made it much more bearable, even bordering on fun at times. She also functioned as a "single parent" much more often than she ever thought she would. My two daughters, Tiffanie and Aimee, likewise put up with a "crabby daddy" alot, but somehow with their own special magic helped keep me sane.

I love all three of you very much. Thank you.

Mark W. Pfefferman

Table of Contents

List of Figures

AFIT/GCS/ENG/GCS93D-18

Abstract

Current dynamic world situations demand flexible training methods which can be rapidly adjusted to vastly different scenarios and conditions. Combat simulations are being aggressively researched by the US military to provide this flexible training capability. While the simulation models become increasingly sophisticated, the basic method of generating the scenario for the simulation is still a manual, error-prone process.

This thesis examines this problem and presents a prototype architecture for an automated scenario generation system. This architecture is designed using an object-oriented approach which leads to a modular and modifiable design. The design allows more sophisticated modules to easily and efficiently replace less sophisticated modules.

The architecture provides a mechanism for automatically generating scenario files from a textual operation order. The format of the operation order is the five paragraph order in use in the US Army. The process is broken into two phases. The first phase translates the operation order into an intermediate format called the mission file. In the second phase, the system reads the mission file, instantiates intelligent entities and assigns missions to those entities. The intelligent entities emulate the subordinate leaders in planning missions. The prototype system designed in this thesis effort assumes that sufficiently sophisticated natural language processing algorithms can translate the operation order into the mission file. The prototype system reads the mission file, conducts mission planning and creates a scenario file for the BATTLESIM simulation developed at the Air Force Institute of Technology.

A Prototype Architecture for an Automated Scenario Generation System for Combat Simulations

I. Introduction

In today's world of shrinking budgets and diminishing enemy threats, the United States military is looking for alternative ways to conduct both analysis and training. Due to the downfall of the Soviet Union and the end of the Cold War, large scale military operations, such as Return of Forces to Germany (REFORGER), are no longer as economically and politically viable as they once were (13:21). Combat simulations are being aggressively researched and developed to provide the necessary training and analysis capability. As Hartman observed,

> For obvious political and financial reasons, it is often impossible to study the behavior of military systems in combat by observing the real system in action... Thus, we study the behavior of models of military systems...(10:1)

As this observation illustrates, combat models have great potential for assisting in military readiness. Modeling, however, is not yet a perfect science. Because of this deficiency, a large number of models have been developed and continue to be developed by the military, National Laboratory System, private industry and academic institutions. One academic institution, the Air Force Institute of Technology, is conducting extensive research in making combat models run faster and more efficiently. One of the main thrusts of this research is the parallelization of combat simulations. Instead of running the simulation on a single processor, parallelization allows the simulation to be divided into independent pieces which can then be executed concurrently on several processors. This capability promises to speed up and improve large scale simulations (11).

1.1 Background

1.1.1 Types of Combat Simulations. A multitude of combat models have been and continue to be developed. In spite of the large number of models, however, the models can be divided into two basic classes: training and analysis. This division recognizes the unique purpose of each class of model.

1.1.1.1 Training Models. The objective of training is to enable personnel to respond appropriately to given situations. The purpose of "wargames" is to achieve this objective. Training models are the most common class of "wargames". The purpose of the training model is to create a realistic and reasonable simulation of actual combat conditions, in order to train personnel.

Because the purpose of training is improvement in human reactions, training models generally have a heavy dependence on player interaction. Player actions drive the conduct of the simulation. Lesser emphasis is placed on the mathematical representations of weapon systems, probability of kill (PK) tables and vehicle characteristics. Greater emphasis is placed on player and enemy actions/reactions to achieve desired training goals. Because of this emphasis, the data for the various systems can be easily changed to give the enemy or friendly side advantages in weapons' capabilities or probability of kill (PK). For example, if the player happens to be an extremely talented and knowledgeable decision-maker, certain values could be adjusted to give the enemy forces an advantage which would challenge him at an appropriately higher level. The training models are primarily used as teaching tools to guide personnel towards a particular training objective, and should, therefore, be relatively easy to adjust to satisfy various training requirements. Training models are not generally used for analysis for two reasons. First, many weapon systems and other systems are not modeled with sufficient accuracy to avoid erroneous conclusions to be drawn concerning new weapons or armor against a given enemy. Second, these types of models are heavily dependent on human interaction. Since human beings do not always react in exactly the same manner in a given situation, the "man-in-the-loop" can also cause inconsistent or erroneous results (16).

1.1.1.2 Analysis Models. In contrast to the training models, analysis models concentrate on simulating weapon systems, tactics and techniques from a more scientific approach. For example, the ballistic properties of each weapon system might be completely and carefully modeled. This type of model is used by analysts to determine the changes which may be introduced to future battlefields, such as new and improved weapon systems or improved protective armor. Generally little to no player interaction is involved. The data for every entity is loaded into the simulation and processed in multiple batch mode runs. Variation is introduced through some stochastic or deterministic process. The results of multiple runs may be analyzed to determine the effects of the new weapon systems or armor. Analysis models are used to analyze scientifically the effects of new entities or capabilities on the future battlefield. Two significant features distinguish this type of model from training models. First, in designing analysis models, the major thrust of the effort is in modeling the physical aspects of the entity (vehicle, weapon system) as scientifically or mathematically complete as possible. Second, because the purpose of the model is to mathematically model the characteristics of an entity, very little emphasis is placed on the actual actions and reactions of the player. In other words, little to no user interaction is involved in this type of simulation (16).

1.1.2 BATTLESIM Model. The Air Force Institute of Technology has developed its own combat model, *Battle Simulation Model* (BATTLESIM). This model is used by several research working groups, most notably in the area of parallel simulations and parallel algorithms. The BATTLESIM Model has been designed to execute on an Intel iPSC/2 Hypercube computer, an eight node parallel processing computer. BATTLESIM most closely fits the analysis model since it requires no user interaction after execution starts, but it differs somewhat in that it is used primarily to test new parallel processing techniques.

1.1.3 Scenario File. Common to all combat models is the need to design a scenario for the model to follow. The scenario specifies all characteristics of the simulation session, such as the geographic area, players involved, and capabilities of the players. The AFIT BATTLESIM combat model requires a scenario file describing the boundaries of the

battlefield, the sectors of the battlefield, the entities in each sector, the weapons of each entity, the sensor ranges of each entity and every route point of each entity's route (2:20).

Many combat simulations use a similar type of scenario file. Currently, creating these scenario files is almost exclusively done by hand. This hand scripting process is slow, tedious and error-prone. Additionally, changes to the original plan, such as the addition of more forces or changes in weapons' capabilities, can often cause extensive changes to the scenario file.

1.1.4 Operation Order. One common element of all military operations, whether simulated or actual live exercise, is an operation order which details who is involved and what exactly the participants are expected to accomplish.

No simulation found in currently available literature has been capable of creating the scenario file directly from the operation order automatically. Several simulations systems have tried to improve the user interface, allowing the user to more easily build a scenario. *Eagle*, a battlefield simulation under development by the United States Army's Training and Doctrine Command (TRADOC), prompts the user through a series of questions. The interface is structured so that the user can extract the information directly from the operation order (17:10). This approach, while an improvement, is still labor-intensive and error-prone.

1.2 Specific Research Problem

The objective of this research effort is to define an architecture for an automated scenario generation system capable of analyzing an operation order, terrain, and doctrine to create scenario files capable of supporting large scale combat simulations. The target system is the AFIT BATTLESIM model.

1.3 Applications for Automated Scenario Generation

The possible applications of automated scenario generation are numerous. By generating large-scale simulations for AFIT's BATTLESIM, researchers can rapidly and efficiently test multiple partitioning strategies and other design aspects of parallel simulation

programs. The process of writing a new scenario to test a particular partitioning strategy, for example, can be reduced from days or weeks to minutes.

Another possible application of automated scenario generation is as a training and evaluation tool for officers learning basic tactics and doctrine at the service schools. For example, a second lieutenant studying the armor platoon in the attack could be given a situation; he must prepare his platoon for a movement to contact, crossing the line of departure at 1700 hours. He has two hours from the time of receipt of his instructions to "issue" his subordinates a five paragraph operation order (by keying it into an ASCII file). The scenario is generated from his plan and discrepancies in his operation order are noted. The young officer's plan is then executed against a simulated enemy force and evaluated for tactical feasibility.

1.4 Assumptions

Several key assumptions were made in this research effort.

- No significant modifications to the current BATTLESIM scenario file format are necessary.
- Sufficiently sophisticated natural language processing systems can be developed to transform a text operation order into an intermediate format to be processed by the scenario generation system. The structured format of the operation order will contribute greatly to this success.
- Terrain files can be generated by a separate automated function. Either data from the Defense Mapping Agency, US Army Corps of Engineers Waterways Experiment Station or created by the user can be used to create a terrain database in a format easily accessible by the scenario generation system.
- Terrain can be divided into traversable ("go") and non-traversable ("no-go") areas. The terrain trafficability is considered to be a binary condition either traversable or non-traversable.
- Obstacles ("no-go" areas) are convex in shape. The obstacles do not have any concave sides. While this assumption is not necessary for automated scenario generation in

general, it assists in obstacle avoidance in the route planner used in the prototype described in this thesis.

1.5 Scope

Although a complete automated scenario generation system would support a wide variety of military missions and entity types, this system concentrates on two missions: "route recon" and "bomb target". The five entity types supported in BATTLESIM (F18, MIG1, missile, tank, and truck) are supported. The missions are read from a mission file, entities are instantiated and the mission is passed to the proper entity for execution. These missions are planned using current military doctrine and control measures. The system will avoid any number of obstacles or "no-go" areas. The output of the system will be a scenario file in the format accepted by the AFIT BATTLESIM model.

1.6 Desired End Results

The end result of this effort is the design of a flexible architecture for an automated scenario generation system which will support reading textual operation orders and creating scenario files with minimal user interaction; and the development of a prototype system which provides proof of concept for possible follow-on research in this area.

1.7 Approach/Methodology

This research effort is presented in three phases. First, the information typically required for scenario files is examined. This phase consists of four tasks:

- Analysis of current scenario generation methods,
- Analysis of the information typically required by a combat model,
- Analysis of any additional information required by BATTLESIM, and
- Determination of the source of the information, such as operation order or doctrine.

The second phase defines an automated scenario generation system capable of reading the required information from the operation order, converting the text information to an

intermediate format, considering doctrine and terrain information, and generating scenario files to support any size combat simulation.

The final phase designs a prototype scenario generation system. Several key subtasks are accomplished in successfully completing this phase:

- Design the prototype model,
- Design a suitable terrain representation,
- Design a suitable intermediate format for the mission file,
- Design a simple mission planning mechanism, and
- Test generated scenario files for compatibility with BATTLESIM.

Since numerous battlefield simulation models exist, each having its own specific requirements, no attempt is made to build a scenario generator capable of producing several formats depending on the simulation model. However, through exploring the requirements for automated scenario generation, this thesis describes an architecture for automating the procedure. By observing the results of a prototype scenario generation system and by testing the scenario files in an actual simulation, the idea of automated scenario generation is proven to be not only valid, but also of great potential value.

II. Literature Review

2.1 Scenario Generation

A wide variety of combat models have been and continue to be developed, most operating under different conditions and assumptions. These conditions and assumptions are commonly referred to as the "scenario". The scenario determines the enemy forces, friendly forces, battlefield size, geographic area, and weapon capabilities that are present on the battlefield. The scenario sets the initial conditions of the combat area and attempts to capture the original assumptions of the combatants prior to the start of the simulation (11:2). The task of entering this scenario information into a combat simulation is currently done almost exclusively by hand. This hand-scripting process is slow, tedious and error prone (22:243).

Depending on the combat simulation model, the amount of information needed for an accurate scenario can take weeks or longer to gather, analyze and enter. This could be a critical bottleneck, especially if the simulation will be used to train forces for contingency missions or to analyze dynamic, changing world situations. Because predicting the next area of the world where U.S. forces will be committed in military operations cannot be done with certainty, analysis and training models must be flexible enough to generate scenarios for the full spectrum of conflict. Additionally, the military may not have the luxury of months of preparation time for the next conflict, as was the case in Desert Shield and Desert Storm (24:15). Thus, a faster and better method for generating scenarios for combat simulations is needed in the simulation world.

The scenario can be created in several ways. First, the scenario can be "hard-coded " into the simulation model. This method limits the flexibility of the simulation to representing only those situations coded into the simulation. Video games use this type of scenario creation. Second, the scenario can be read from a file or database. This method allows scenarios to be developed from a set of possible terrain, enemy forces, friendly forces and weather parameters. Another possible implementation of this is to store a set of complete scenarios. This method provides additional flexibility, but can require large amounts of storage. The *Rasputin* system requires over 90 gigabytes of mass storage for

the scenario, terrain and entity files (18). Third, the scenario can be computer generated. This emerging technology provides flexibility in scenario creation by allowing scenarios to be generated from an operation order. This type of system can be divided into semi-automated and fully automated systems.

2.1.1 Semi-Automated Scenario Generation Systems. This category of scenario generation systems provides a much improved user interface to the model. Work has been conducted by the US Army's Training and Doctrine Command's Analysis Command, Los Alamos National Laboratory and the Naval Postgraduate School in creating a more flexible and powerful scenario creation procedure. The results of two efforts, *Eagle* and *CASTFOREM*, will be examined more closely.

2.1.1.1 Eagle. The US Army's *Eagle* battle simulation is one of the few models with automated scenario generation features. *Eagle* provides a sophisticated user interface called the Intelligent Plan Preprocessor (IPP). The IPP allows the user to build the scenario using information taken straight from the operation order. The orders are automatically sent to lower units for implementation (6:4). This method, while a significant advancement from the previous hand scripting methods, still requires an analyst or team of analysts to define terrain, mission and decision rules. Changes in unit mission, unit locations, and unit strengths can cause a substantial amount of reworking and debugging (22:244).

2.1.1.2 CASTFOREM. The US Army's *Combined Arms and Support Task Force Evaluation Model (CASTFOREM)* is a Corps level combat simulation which requires a great deal of set-up time and effort. The Scenario Writer's Guide identifies a six step process for creating scenarios:

- Step 1.
  - Step 1a. Data search
  - Step 1b. Write ground maneuver data sets
  - Step 1c. Line of sight analysis for unit positions
  - Step 1d. Develop routes and battle positions

- Step 2.
  - Step 2a. Create combat orders
  - Step 2b. Model command procedures and audits
- Step 3.
  - Step 3a. Formations and communications
  - Step 3b. Decision tables for movement
  - Step 3c. Movement trial runs
- Step 4.
  - Step 4a. Search areas of responsibility
  - Step 4b. Surveillance decision tables
  - Step 4c. Surveillance trial runs
- Step 5.
  - Step 5a. Repeat steps 1 to 4 for:
    - Artillery
    - Engineers
    - Helicopters/Air
    - Air Defense
  - Step 5b. Trial runs for each addition
- Step 6.
  - Technical data scrub
  - Replications

The Scenario Writer's Guide states that much of this information is provided in the form of existing data sets (12:9–11). Step 5a, however, is to repeat all previous steps at least four more times. This procedure, while providing the user a great deal of flexibility, could be a long, tedious and error-prone process. Like *Eagle*, changes to the original set of assumptions can cause substantial re-working of the entire scenario file.

2.1.2 Fully-Automated Scenario Generation Systems. This research area is new and no literature describing a fully automated system is currently available. The basic premise for this technique is that scenarios can be automatically created from the military operation order. Regardless of the type of operation or exercise, some form of a military

operation order is published detailing the participants and their missions. The fully automated scenario generation system considers the mission, enemy and friendly doctrine, terrain and the weather to create a realistic and complete scenario. To accomplish this task, the system must possess two capabilities. First, the system must "read and understand" a text order. Second, the system must "know" how to conduct certain missions in order to create an accurate and realistic scenario file. Artificial intelligence techniques, such as natural language processing and intelligent mission planning, could provide automated scenario generation systems with these capabilities.

2.2 Natural Language Processing

Natural Language Processing involves a wide range of disciplines ranging from computer science to psychology. The thrust of natural language processing is to make the interface to a computer easier through the use of human language, either spoken or written (14:1242). In this thesis, only written language, referred to as text, is considered.

Natural Language Processing is considered by many computer scientists to be a problem which is not solvable in polynomial time (NP-Complete). NP-Complete problems are a category of problems which have a combinatorial explosion as the number of inputs increases. A text processing system for the English language, for example, must have the capability to understand words, phrases and sentences which can have vastly different meanings when considered in different contexts. To consider all the possibilities is a nearly impossible task (19:377-379).

The natural language processing task can be simplified by creating the operation order using only a small subset of English. The majority of the terms in an operation order are commonly used military terms, such as "movement to contact", which usually have a very specific, unambiguous meaning (5:5–2).

2.3 Operation Order

The operation order gives exact details concerning a pending operation. The commander of an operation uses it to communicate his plan to his subordinate unit leaders.

Upon receipt of the operation order, the subordinate unit leaders plan their portion of the operation and issue an operation order to their subordinates.

Figure 1 shows a sample format for an operation order commonly used in the United States Army. The five paragraphs, Situation, Mission, Execution, Service Support and Command and Signal are always found in an operation order, regardless of the size of the organization. These sections detail specific information related to the particular operation described in the order.

```
 I.  Situation
II.  Mission
III. Execution
     a. Concept of Operation
     b. Tasks for Subordinate units
     c. Coordinating Instructions
 IV. Service Support
  V. Command and Signal
     a. Command
        1. Commander's location
        2. Command Posts Location
     b. Signal
        1. SOI edition
        2. Challenge/Reply
```

Figure 1. Sample Operation Order Format

2.3.1 Situation. This paragraph gives details of forces, enemy and friendly, in the operational area. This information is arranged in three subparagraphs:

- Enemy Forces. Includes information on enemy locations, weaknesses, strengths and recent activity.
- Friendly Forces. Includes information on friendly units operating to the left, right and rear.
- Attachments and Detachments. Lists any subordinate units attached or detached for the duration of the operation.

2.3.2 Mission. The task that is to be accomplished is given in a clear, succinct statement. This statement will include what, who, when, and where.

2.3.3 Execution. The purpose of this paragraph is to explain how the mission is to be accomplished. This paragraph is arranged into three subparagraphs:

- Concept of Operation. Includes scheme of maneuver, obstacle plan, and any other clarifying information that the commander considers critical to his subordinate commanders.
- Tasks for Subordinate Units. One paragraph is included detailing specific tasks for each subordinate unit.
- Coordinating Instructions. Includes any tactical instructions applicable to two or more subordinate units.

2.3.4 Service Support. This paragraph contains information pertaining to rations, medical support, ammunition, transportation, and any other logistical support.

2.3.5 Command and Signal. This section of the operation order contains subparagraphs pertaining to the chain of command, location of the commander, location of the command posts, call signs, radio frequencies and other communications instructions in effect during the operation.

2.3.6 Appendices and Annexes. In addition to the standard five paragraphs, a number of appendices and annexes can be attached to operation order. These attachments give further descriptions of specific areas. The Intelligence Annex, for example, is a highly detailed analysis of the battlefield geography and topology. Key terrain, obstacles, roads, bridges, enemy positions, meteorological data and civilian information are carefully and completely identified in this section (4).

The format and terminology used in the operation order is recognized by nearly all US Army Officers. It is an area of training common to all Army Officers and is used to communicate, unambiguously, missions and requirements for all levels of command

from theater level down to squad level. This specialization of terms, commonality of use, and physical partitioning of the information into subparagraphs greatly facilitates human understanding. Each item of information has a precise location within the operation order. Additionally, the contents of each subparagraph are well specified. This structured format greatly reduces the complexity of the process of extracting key data from the operation order. This is the basis for the assumption that the text operation order can be transformed into an intermediate format through the use of natural language processing.

2.4 Operational Considerations

The operation order is not the only tool used by military planners in mission planning. Many military planners also use "METT-T" to establish a framework to plan their operation. The acronym stands for Mission, Enemy, Terrain, Troops available and Time available (4:2-3). METT-T is a common tool for many military planners in determining operational details.

2.4.1 Mission. The mission defines what the participants in the operation are expected to accomplish. The mission statement is given in clear, unambiguous terms. This mission statement while clear and unambiguous, often requires *implied* tasks to be accomplished as well as the *stated* tasks. An example stated task is: "Company A conducts a route recon mission from Point A to Point B, commencing 120300z September 93." The *implied* tasks might be: "top off all vehicles with fuel prior to departing and ensure that all participants have all their required equipment."

2.4.2 Enemy. The enemy situation is a critical consideration in mission planning. The military planner must know as much about the enemy as possible. Enemy locations, strengths, weaknesses, weapons, unit and morale are all carefully considered when planning an operation.

2.4.3 Terrain. The terrain over which the operation will be conducted is a critical consideration. Military planners as far back as Sun Tzu have recognized the importance of knowing the operational area (25:50). Similar to the acronym of METT-T, another

acronym OCOKA is often used by military planners to assist in the critical task of terrain analysis. The OCOKA acronym stands for:

- Observation and fields of fire. This aspect is considered when placing weapons systems to cover likely enemy paths.
- Cover and concealment. This aspect influences choice of routes and positions. Both routes and positions are chosen to maximize cover and concealment.
- Obstacles. This aspect also influences routes and positions. Routes are generally chosen to avoid obstacles, while positions generally are chosen to integrate obstacles.
- Key terrain. This aspect describes any terrain feature which if held by either side gives that side a marked advantage. Both routes and positions are chosen to maximize the use of key terrain.
- Avenues of approach. The other four aspects are considered in conjunction to determine the likely paths of enemy movement and best routes for friendly movement (4:2-3).

2.4.4 Troops Available. This aspect of military planning defines the friendly forces available for the mission. Units are often attached and detached to support another unit in different missions.

2.4.5 Time Available. This aspect of METT-T tells the military planner how long from receipt of an operation order he has to plan his actions. Usually this is given in the form: "Company A will conduct a Route Recon commencing at 122300z August 93." The planner figures the difference between the time that he receives the order and the first action time in the order. That difference is the time available for the planner to formulate a plan and to put it into action.

2.5 Intelligent Mission Planning

To emulate a subordinate unit leader planning his mission, an automated scenario generation system needs some type of intelligent mission planner to determine the actions of

the forces in the simulation. The operation order gives missions and objectives and allows the subunit leaders to determine the best way to accomplish the given mission. In other words, the operation order tells when and what is to be accomplished, but the subordinate leaders determine how a mission will be executed. The automated scenario generation system will emulate this subordinate planner by conducting the mission planning.

Two general approaches to mission planning can be used. The first method is to design an overall simulation control module. This module would perform all mission planning, resolve conflicts and pass carefully constructed scripts to the entities to execute. The entities would then not have to be very intelligent or sophisticated since the detailed planning and re-planning would conducted at a higher level.

The second method is to make the entities the intelligent portion and allow them the freedom to plan their own missions when given directives from the simulation. This method is similar to the process of human military planning. The difficulty in this method is that the entities need to know about the simulation environment and about the other entities in that environment. Additionally, some method of conflict resolution is still needed at a higher level to handle problems like route de-confliction, coordinated maneuvers, and any other issues caused by object interactions (22:248).

Regardless of the method used some type of planning mechanism must be designed to perform mission planning. This planning mechanism should closely emulate the planning cycle that a subordinate leader uses. One example of a computer emulating a subordinate planner can be seen in research being conducted by the Loral Systems Company in "Semi-Autonomous Forces" to be used in the Distributed Interactive Simulation system. These forces will be capable of realistic independent action. The idea is for these forces to simulate both subordinate units and enemy units. This capability allows relatively large exercises with maximum leader involvement and minimal soldier deployment (8:21–50). In a similar manner, a "semi-autonomous" entity could be used to plan missions, resulting in the creation of the scenario file.

Another example of research in Intelligent Mission Planning is the Pilot's Associate Project. The goal of this project is to develop an intelligent on-board computer assistant

for a pilot. This system would relieve the pilot of some routine tasks. Kilpatrick's work in discovery-based learning demonstrated that an onboard system could learn to avoid certain radar sites. More importantly, Kilpatrick's *Maverick* system improved as it discovered more efficient methods to avoid certain radar sites (15:50-75).

The manner in which the techniques used in these systems can be leveraged will be presented later in this thesis.

2.6 Terrain Representation

Any intelligent mission planner must interact with terrain features. These terrain features can be represented in several ways. Two terrain representation methods are presented here.

First, the terrain features can be described by sets of line segments, which trace the surface of the particular feature. This method can produce polyhedra representations of terrain features. This representation produces objects which accurately portray reality. This representation has been useful in a wide variety of domains, such as circuit board layout (1:285).

Second, the terrain can be divided into uniform polygons, such as triangles, hexagons or squares. Each of the polygons can then be treated as a separate object with its own associated attributes, such as vegetation, height and elevation. Hexagons have been used successfully in many "wargame" applications. Any uniform discretization of terrain, however, regardless of the shape and size of the polygon chosen, causes some deviation from reality. The real world does not generally neatly divide into uniform size pieces. Additionally, contouring or resolution is lost since the area within the polygon becomes a single data point. Gentle slopes can become steep steps between adjacent polygons. One advantage of this method is that the Defense Mapping Agency and the U.S. Army Engineer Waterways Experiment Station have data on many areas of the world in this type of format (20:6).

These are two methods which are used to represent terrain in digital form. The exact method implemented in this thesis effort is presented in Chapter 4.

2.7 Conclusion

Most of the work in combat simulations to this point seems to have concentrated on improving the inner workings of the model. Because of this concentrated focus, a large number of models were developed to meet specific requirements. Improving the scenario generation process for combat simulations appears to have been a lower priority for researchers. This observation is based on the lack of literature currently available on automated scenario generation systems. Consequently, the process of creating new scenarios has been oftentimes slow and difficult. Systems are being developed which give the user greater flexibility in building scenarios (12) and have a better user interface for building scenarios (17). This thesis effort concentrates on further improvement in this area by fully automating this process. Fully-automated systems capable of reading text orders, of emulating the subordinate planner conducting his assigned missions, and of creating a properly formatted scenario file can provide the military with an efficient and effective support tool. Current research in the areas of natural language processing and intelligent mission planning applications appear to be potential solutions for providing these capabilities in an automated scenario generation system.

III. Methodology

3.1 Introduction

This chapter describes the design issues in defining an architecture for an automated scenario generation system and then proposes the implementation of a prototype to demonstrate the validity and utility of this design.

3.2 Approach

The Object-Oriented Design and Programming paradigm was chosen as the design methodology. The advantages and disadvantages of an object-oriented approach versus other methods have been well documented by Rumbaugh and others (21:266–274). According to these authors, the advantages of an object-oriented approach include:

- Object-oriented approaches concentrate on real world objects as they exist in the problem domain (21:4–5).
- Object-oriented designs are based on the underlying structure of the problem domain and not on a designer imposed decomposition (21:7).
- Implementation issues, which may be difficult to identify in the initial stages of a design project, are deferred until implementation. This helps to prevent early implementation decisions which result in too restrictive a design and an inferior product (21:4).

The design which results from using an object-oriented approach is more flexible to changing requirements and more closely matches the problem domain (21:4–7). Because of these advantages, this thesis uses object-oriented methods to design the automated scenario generation system architecture.

3.3 Automated Scenario Generation

Figure 2 illustrates the overall scenario generation process. The input to the system is the operation order. This order is processed by the text processing system and results in an intermediate format (independent of the target simulation) called the mission file. The

mission planner reads the mission file, instantiates the proper objects, and assigns them their respective missions. The objects, then acting as "subordinate leaders", plan their respective missions. Each entity completes its mission planning and passes its plan back to the mission planner. The mission planner then writes these results to a scenario file in the proper format for the target simulation system. The completed scenario file can then be given to the target simulation system for execution.

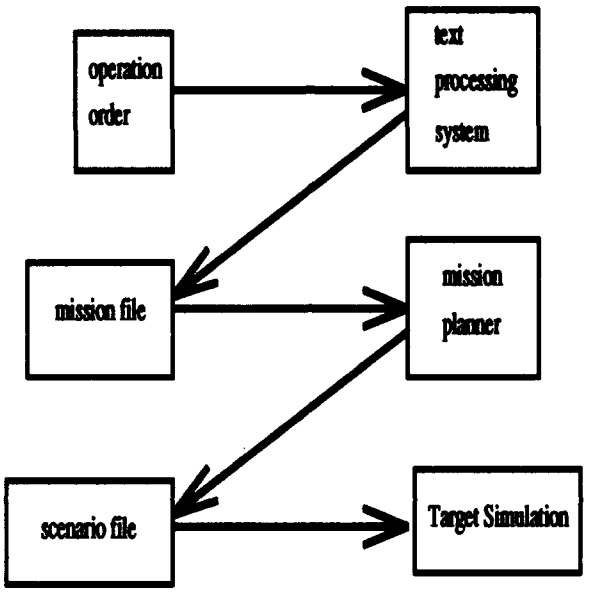

Figure 2. The Scenario Generation Process

In order to create a system which is as flexible as possible, the system should consist of replaceable components. As more sophisticated methods are developed, these new modules can be easily and seamlessly substituted for the old modules.

3.4 Operational Considerations and the System Architecture

The automated scenario generation system is designed to emulate the actions of the subordinate unit commander in interpreting the operation order and planning his portion of the mission. Therefore, the automated planning algorithm should consider the same decision making criteria and approach used by the subordinate planner upon receipt of an operations order from his commander. In planning military operations, many military planners use the acronym: "METT-T" to establish a framework to plan their operation.

This acronym is defined in Chapter 2. In this chapter, the manner in which the automated scenario generation system represents METT-T considerations is examined.

3.4.1 Mission Representation. The mission defines what the participants in the operation are expected to accomplish. The mission statement is given in clear, unambiguous terms.

This mission can be read directly from the operation order and transformed into an intermediate format which is independent of the target system. The intermediate format, stored in the mission file, gives the key information which is needed by the intelligent mission planner. As a minimum, this file should contain entity name, entity type, mission, date/time group and any key route points. Each piece of information required in the intermediate format can be traced directly back to a paragraph in the operation order.

3.4.2 Enemy Representation. The enemy situation is a critical consideration in mission planning. The automated scenario generation system considers the enemy locations, strengths, weaknesses and unit when planning the steps to complete its mission. If the mission is to "move to point A avoiding enemy detection", then the automated scenario generation system should plan a route which avoids all suspected enemy locations. Conversely, an attack mission might be planned to move straight to a particular enemy location and to engage any enemy forces encountered along the route.

In representing enemy forces, some identifying feature, such as "nationality" or "loyalty" can be included in the entity and set to different values for enemy forces. Additionally, in a graphical display, different forces can be displayed using different colors schemes, shapes or symbols to identify different forces involved in a conflict.

3.4.3 Terrain Representation. The terrain over which the operation will be conducted is a critical consideration. Maps, photographs and reconnaissance are all used by the military planner. Computers, however, have a limited ability to use these intelligence products in the same manner as the human planner. Because of this limitation, some form of a digitized map must be developed and entered into the computer.

The Defense Mapping Agency (DMA) can provide digitized maps for many areas of the world. The resolution of these maps can be as small as 1 arc second or roughly 25 meter squares. This terrain description, however, only consists of longitude, latitude and elevation information. Other agencies can provide limited vegetation and other feature information (20:10).

The terrain "file" can be in several different forms. For example, the *Rasputin* system uses relational databases to store terrain information (18). Another storage method uses ASCII files with records of uniform length to hold terrain information. Any number of methods can be used to represent and store terrain data for the operational area. Each method has its own associated advantages and disadvantages.

3.4.4 Troops Available Representation. This aspect of military planning defines the friendly forces available for the mission. An automated scenario generation system may typically represent these forces as lists, arrays, or objects. Under the object-oriented paradigm, these entities should be objects. Each object has an identity which is unique to that object, special slots called attributes to hold data values and a set of object specific functions called operations. Attributes are data values only and are not required to be unique among objects. That is, multiple unique objects can have the same attribute values. Operations are functions which can be used to access attributes in an object or actions which the object can perform (21:22–26). Attributes and operations are generally specific to a class of objects and can be passed down from the object class or type to the specific instance of the class. A truck, for example, could be a subclass of the vehicle class with the attributes of tires and a body color. The operations for the vehicle class could be *drive* and *break down*. A particular green High Mobility, Multi-Purpose Wheeled Vehicle (HMMWV) would be an instance of the truck subclass and would inherit the operations of *drive* and *break down*.

Entities in an automated scenario generation system can likewise use the concepts of classes to define types of vehicles. Tracked vehicles may be a subclass of the category of vehicles or a distinct class by themselves if the system designer decides that tracked vehicles are not similar to wheeled vehicles. This modification would require redefining

the classes in the entity data base. As long as the interface between the planner and the entity stayed the same, no changes would have to be made in the planning code.

Using BATTLESIM as an example, the following attributes and operations fully describe an entity used in a simulation session:

- nomenclature
- type
- id
- speed
- fuel_capacity
- mobility
- min_path_width
- max_climbing_height
- max_ford_depth
- sensors
- sensor_type
- sensor_range
- sensor_resolution
- armaments
- armament_description
- defensive_systems
- defensive_sys_description
- loyalty
- fuel_status
- condition
- vulnerability
- x_velocity
- y_velocity
- z_velocity
- yaw_rate
- pitch_rate
- roll_rate
- experience

- threat_knowledge
- min_turn_radius
- avg_fuel_consumption
- max_climb
- targets
- target_list

Entities are generally one tank or one truck, but in many combat simulations the players represent aggregate entities, such as platoons or companies. This representation can be handled in this system in two ways. The first method is to create an aggregate object to represent a group of objects. For example, an entity called "tank platoon" can be defined and represented as a completely new object. One aspect of the aggregate object is that it requires a larger area of terrain to maneuver than a single entity does. The proposed attribute of "min_path_width" is larger for a tank platoon than for a single tank. This attribute represents the doctrinal minimum path width for a particular type of entity. Routes for the missions would then be planned with that minimum path width as a consideration. The costs of choosing a path that was less than the doctrinal minimum width could be compared with the costs of possibly longer routes with doctrinally acceptable paths. Cunningham suggests a method for measuring the costs of moving formations of objects through restrictive terrain (3). The second method is to simply create a new object consisting of one or more basic objects. In this representation, a tank platoon would consist of four tanks which move as one entity. Cunningham's cost measuring method can be applied to this aggregate representation as well (3). Either of these two methods could be effectively employed to represent an aggregate entity.

Not only does an object have characteristics, it also has some functions which it can perform. These functions are the military missions that a particular object can perform. For example, each object which can perform a "recon route" mission would have a "recon route" operation or function associated with it. This feature provides a mechanism for eliminating nonsensical operations, such as "bomb target" for a truck object. In an object oriented environment these operations are defined for the object or inherited from the object's class.

3.4.5 Time Available Representation. For this research, the time available to plan is not examined. The thrust of this research effort is to reduce the time to generate large scale scenarios through automated methods and not to design a planner which can generate a plan within some time constraint.

3.5 Final Design

Figure 3 is the object diagram representing the overall structure of a prototype architecture for an automated scenario generation system. Each object has been designed to represent a highly specialized function. This partitioning allows the overall system to be as modifiable as possible.

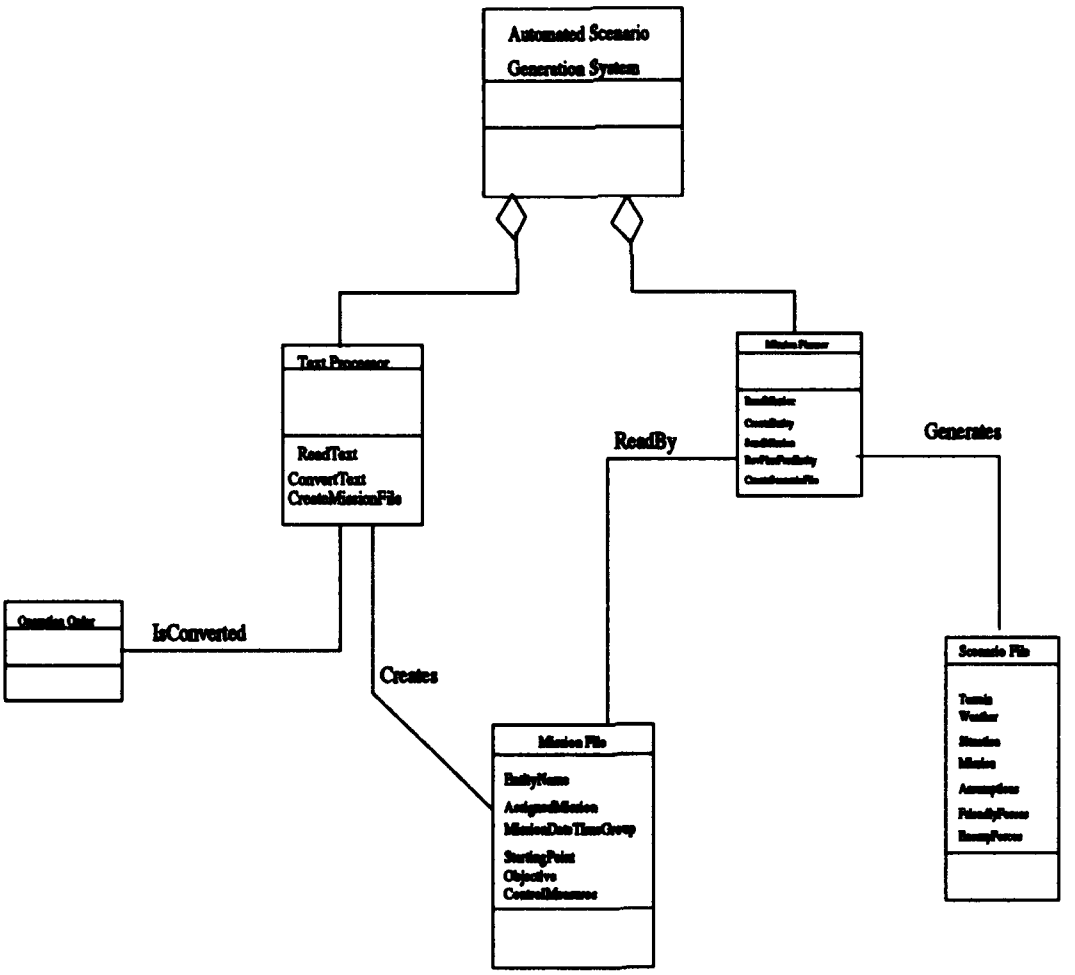

Figure 3. Object Diagram of Overall System

3.5.1 Operation Order. Figure 4 shows the structure of the operation order object. The arrangement and contents of each paragraph of the operation order were explained in Section 2.3.

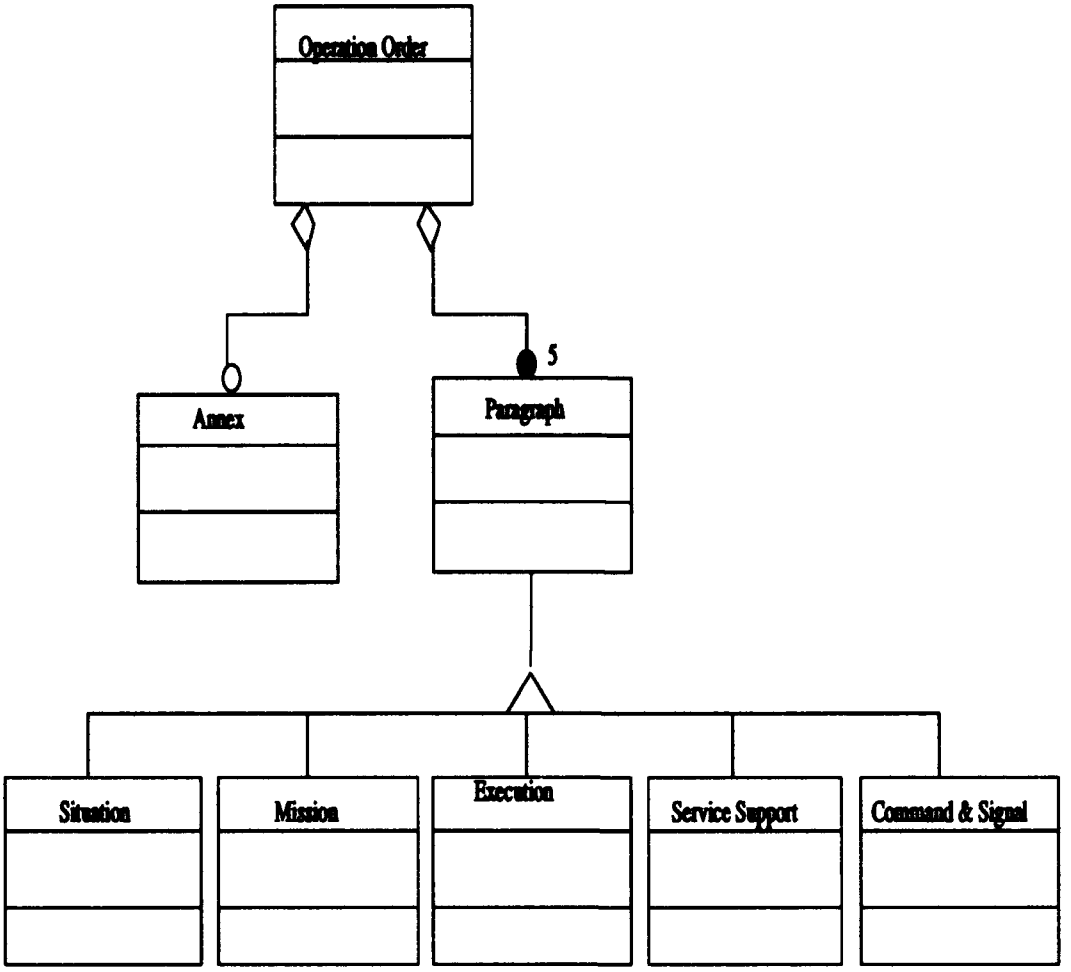

Figure 4. Operation Order Object Structure

3.5.2 Text Processor. This portion of the automated scenario generation system reads in an operation order in text format, parses the text and creates an intermediate format called the mission file. The text processor's task is made easier since the operation order is in a highly structured format and written in precise, unambiguous terms.

3.5.3 Mission File. Paragraph 2, *Mission*, of the operation order gives a succinct statement of the task that is to be accomplished during the operation. This mission

statement uses precise, unambiguous terms to explain the mission. "Third platoon will conduct a hasty defense commencing at 1200 hours to defend key terrain at WP487394 from possible enemy counter attack" is a clear and concise order, easily understood by all competent military officers. The terms "route recon" and "hasty defense" are tasks conducted using specific techniques. These specific mission terms are then easily mapped into the intermediate format. The mission statement above would translate into the following entry in the mission file:

```
1 3PLT INFPLATOON HASTY-DEFENSE 091200z_August_1993 WP48791234 WP50734321
```

where the fields represent, respectively, mission number, unit id, unit type, mission, execution time and mission start point and objective location.

Paragraph 2, *Mission*, not only gives the mission to be accomplished, but also tells when the participants are to start the mission and/or complete the mission. This date/time group can then be directly mapped into the intermediate format.

3.5.4 Mission Planner. Figure 5 shows the key components involved in mission planning. The mission planner reads in the data from the mission file, creates the necessary entities, translates the mission, and then passes the mission to the entities involved in the mission. As the entities complete their planning, they pass the results back to the mission planner. The mission planner is then responsible for writing those results to a scenario file in the format accepted by the target simulation system.

3.5.5 Route Planner. The route planner object provides each entity with a common method for path planning. Different route planning algorithms can be implemented to meet specific mission or doctrinal requirements.

3.5.6 Scenario File. The output of the automated scenario generation system is the scenario file. The format for this file is target system dependent. The text processing system of an automated scenario generation system may extract information about geographic location, friendly and enemy locations, friendly and enemy capabilities and other administrative information directly from the operation order and write it directly to a

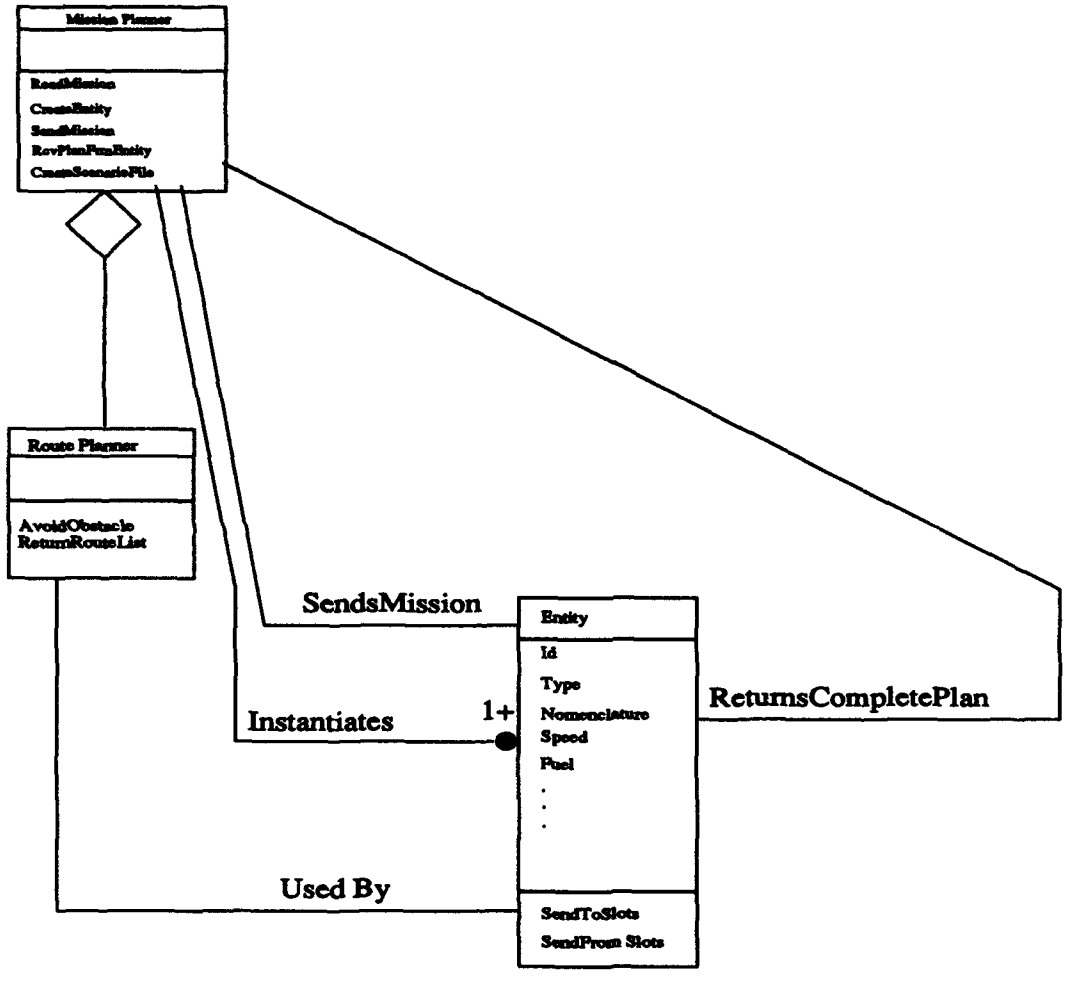

Figure 5. Key Components involved in mission planning

corresponding field in the scenario file. The details of the mission planning, however, are created by the intelligent entities. The intelligent entities are instantiated by the mission planner and given a mission to plan. The results of that planning are then passed back to the mission planner for recording in the scenario file. The scenario file format can range from relational databases to simple ASCII text files (2)(18). Appendix A is an example BATTLESIM scenario file which uses an ASCII file.

3.5.7 Entity. Figure 6 shows the structure of the entity object and the two objects that the entity object interacts with during mission planning. The terrain object represents the geographic and physical arrangement of the battlefield. The doctrine object assists the entity object in making decisions during mission planning. Paragraph

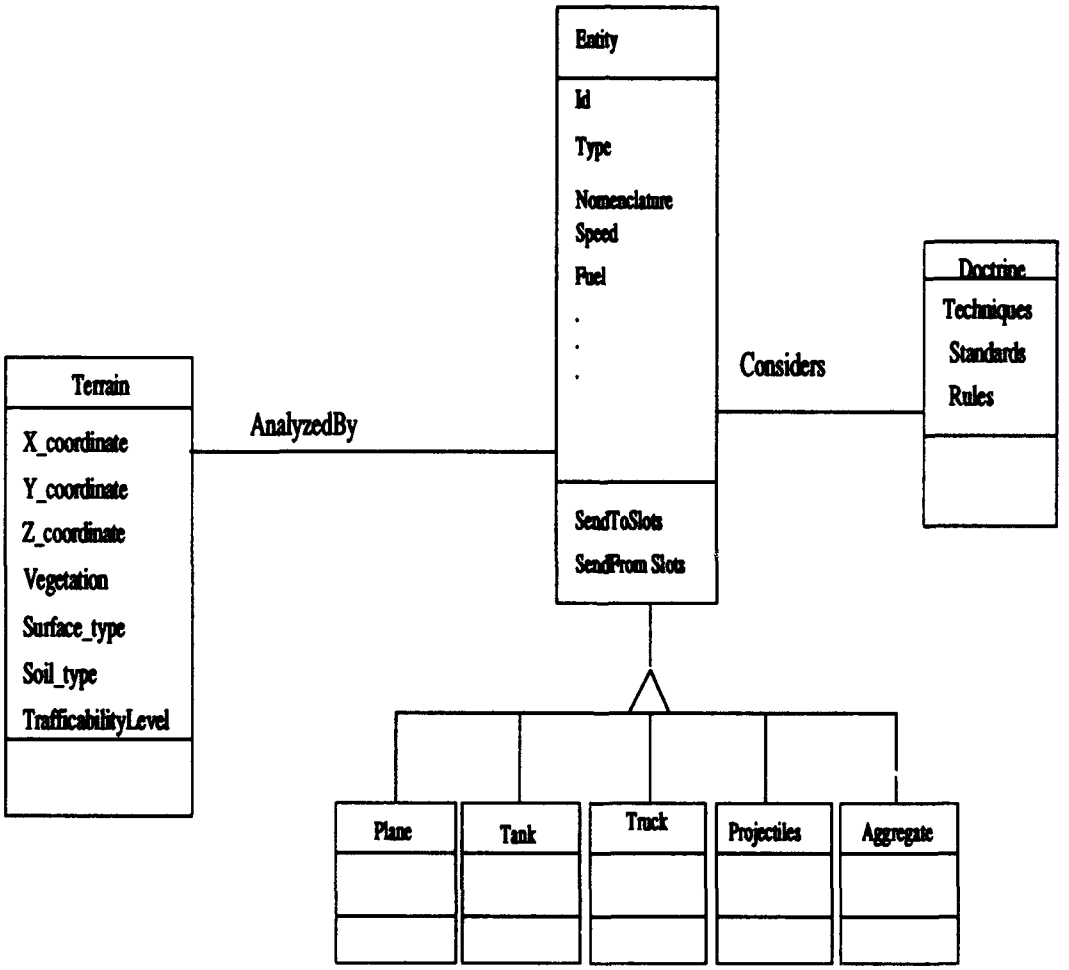

Figure 6. Key Interactive Objects in Scenario Generation

One, *Situation*, of the operation order gives a detailed listing of friendly and enemy forces participating in an operation. This information provides the list of players which can then be verified using later paragraphs. In addition, US Military units generally use the same naming convention of numbered divisions, numbered brigades, numbered battalions, lettered companies, numbered platoons, numbered squads and letter-numbered vehicles. This convention makes determining player names a straight-forward task.

An integral part of the unit name is the type of unit. F Company, 40th Armor Battalion, for example, indicates a unit of approximately 80 soldiers and 12 tanks. Paragraph One, *Situation*, of the operation order would also provide this information, as well as giving the exact type of tank, M60A3 or M1A1, that F Company possessed.

The entities are the "intelligent" portion of the automated scenario generation system. After receiving the mission from the mission planner, the entity executes its mission. In executing missions, the entity will require some form of reasoning ability, such as a rule-base. One possible method is to design the entity as an intelligent object, using the object oriented features of packages, such as TeKnowledge's M4 Expert System (26) or NASA's CLIPS Object Oriented Language COOL (23).

3.5 8 Terrain. This thesis effort represents terrain as a series of 10 meter by 10 meter gird squares with each square possessing an associated mobility factor. This resolution is the same as 1:50000 military maps in use throughout the US Army. This representation assigns each square a unique eight digit identifier which can be used to easily plot routes using simple Euclidean geometry.

These squares are stored as 20 byte strings in a terrain file. The eight digit grid designator, XXXXYYYY, is used as both the grid name and as the record number in the terrain file. The test files consisted of 9,999,999 grid squares ranging from grid 0000,0000 to grid 0999,9999.

This representation has been chosen for several reasons. First, this representation is the same resolution used in 1:50000 military maps which is familiar to most military planners. Figure 7 contains an example map grid. Four digits (XX YY) define a 1000 meter by 1000 meter square area, shown in bold face. Each additional digit defines an

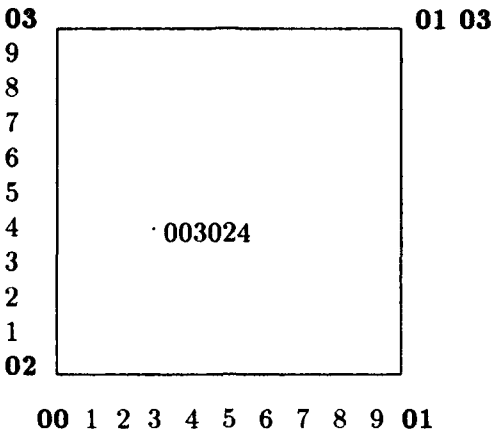

Figure 7. Example grid square, 1000 meters by 1000 meters

increasingly smaller square. Eight digits, XXXXYYYY, define a 10 meter by 10 meter square. This is the resolution used by this system. The smaller numbers are included for illustrative purposes only. These numbers are used to estimate positions within the grid square. These smaller numbers define a 100 meter by 100 meter square or six digit grid location. These numbers are never included on a normal 1:50000 military map. If the space between each smaller number was then divided into ten more segments, then squares of eight digits would be defined and the resolution would be 10 square meters. Second, this representation is easily scalable to larger resolutions. For example, making the rightmost digit "0" in the X and Y component causes the the resolution to be 100 by 100 meters. For example, in Figure 7, the eight digit representation of the highlighted point is 00300240 and the six digit representation is 003024.

Third, distance can be easily calculated using simple Euclidean geometry. The distance between any two points, (x0, y0) and (x1, y1), can be found using:

$$\text{distance} = \sqrt{(x1 - x0)^2 + (y1 - y0)^2}$$

This method of terrain representation is common to military planners and can provide efficient and effective access to terrain information for route planners.

3.5.9 Doctrine. Although Paragraph 3, *Execution*, gives explicit mission definitions, some details of the mission may be missing. These missing details can often be found by comparing the given mission with doctrine. This doctrine object provides the basis of

"how" to conduct a particular mission. The doctrine object can also provide information which is common to many military missions. The doctrine object can provide guidance to the entity in basically two forms. First, the doctrine object can provide previous plans for consideration. These plans can be gathered through experience, pre-loaded based on some historical data or some combination of both sources.

Pre-loaded plans form the basis of information common to all objects. One example of this common type information is a control measure. Control elements of the operation, such phase lines and checkpoints, are given in this paragraph. The presence of these control measures in the *Execution* paragraph indicate a mandatory point that a participant must go through to successfully complete his mission. These control measures are added to the intermediate format after the start and objective points as points which must be traversed on the route or axis of advance.

Plans that have been devised based on experience are usually unique to a particular object until enough experience is gained by all simulation participants to make this information common to all participants. The design of the combat model will determine the location where this information is maintained. Two possible locations are at the simulation level and at the object level. Storing these experienced-based plans at the simulation level assumes that the simulation controls the entities and their planning mechanism. Storing these plans at the object level, on the other hand, assumes that the objects are sophisticated and have knowledge of the simulation environment and the other objects in the simulation.

Each method has its own advantages. Since this architecture concentrates on generating the initial scenario and not on replanning, either method can be supported by this architecture.

3.6 Conclusion

The object-oriented design of the architecture presented in this chapter facilitates straight-forward modification of each object. This feature allows more sophisticated algorithms to be easily implemented for route planning, mission planning and text processing,

as required. The proposed architecture for an Automated Scenario Generation System presented in this chapter provides the flexibility, expandability and versatility necessary to support large scale scenario generation.

IV. Implementation and Testing

4.1 Introduction

Based on the architecture proposed in Chapter 3, a prototype Automated Scenario Generation System was implemented. Figure 8 provides an overview of the prototype system and shows the data flow between the various modules.

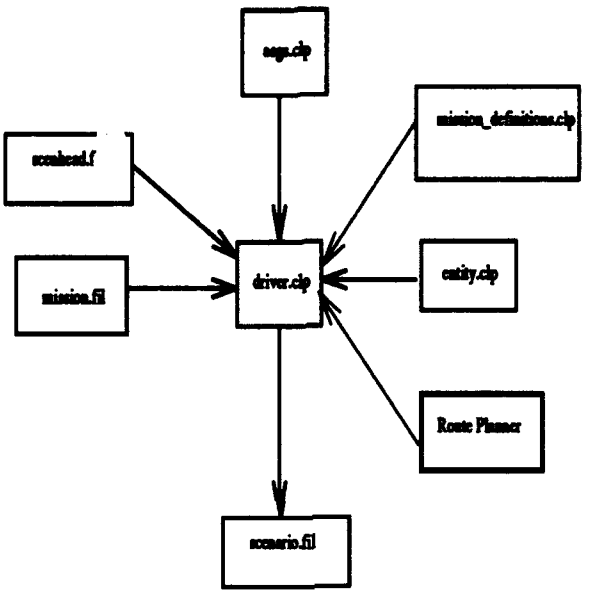

Figure 8. System Overview

4.2 Selection of Language

The C Language Integrated Production System (CLIPS) was chosen to implement this prototype system. CLIPS is an expert system development tool produced by the National Aeronautics and Space Administration's (NASA) Software Technology Branch (STB). CLIPS allows users to build both stand-alone and integrated expert systems.

CLIPS was chosen to implement the prototype system for three reasons. First, in order to create an intelligent entity, the tool chosen had to provide some reasoning capability. CLIPS is an expert system development tool and provides the necessary reasoning ability for a prototype system. Second, the tool to implement the prototype had to have an object oriented capability. The CLIPS Object Oriented Language (COOL) extension provides an

object oriented programming capability, fully supporting abstraction, encapsulation, inheritance, polymorphism, and dynamic binding (23:18–19). Third, CLIPS is designed to be extendible. The user can add new CLIPS commands and functions to the existing CLIPS program. Fourth and most importantly, the prototyping tool had to provide an environment which supported incremental building of the overall system. This capability allows the system designer to add new rules, entities and functions without re-compiling. Rules or constructs which need to be modified can be re-entered at the command line and tested quickly. All the existing rules can be cleared or new rule sets can be written over existing rules. Additionally, rule bases can be loaded, tested and modified incrementally. This capability is not supported in any procedural-based languages to include the interpreted versions. CLIPS provides this environment and these capabilities.

While Lisp environments can generally provide similar features, one additional consideration made CLIPS the more attractive choice. CLIPS provides a built-in inference mechanism based on the Rete Algorithm (23:211). This feature allows users to concentrate on the development of the rule base and not the inferencing mechanism.

Lastly, CLIPS comes with the source code and makefiles, which allow users to add new features to the original CLIPS. This feature is an extremely useful aspect of CLIPS, allowing system designers to tailor CLIPS to their specific application and allows CLIPS to be portable across a wide range of platforms. The key idea in adding new commands to the original CLIPS command set is that any new commands that will be called from the CLIPS command line must be defined in the CLIPS *main.c* file. Any parameters passed to or from those new commands must added to or read from the CLIPS symbol table. Any other routines that are called by the new command routines, but are not called from or pass data back to the CLIPS command line only need to be defined as any normal C routine would be declared.

In this research effort, two features were added. One feature was the direct access routine which provides CLIPS the method to retrieve grid records from the terrain file. The second CLIPS extension *route_planner* is an implementation of Bresenham's Algorithm. This algorithm is used in the computer graphics domain to determine which pixels to illuminate when plotting straight lines between two points (9). The algorithm is used in

this prototype to determine which grid squares are traversed when traveling from a start point to an objective. The details of this process will be presented later in this chapter.

4.3 Automated Scenario Generation System Key Components

The key components of the Automated Scenario Generation System Prototype displayed in Figure 8 are divided into CLIPS program files and data files. The next four sections describe the functions and features of the CLIPS programs in the Automated Scenario Generation System prototype. The three sections which follow the program sections describe the data files which are necessary for the Automated Scenario Generation System prototype.

4.3.1 Route Planner. The route planning mechanism should be selected based on the format chosen for the terrain representation. The terrain representation can be broken into "go" areas and "no-go" areas based on strictly on trafficability considerations. Then by considering enemy positions and capabilities, such as enemy weapons' placements, more "no-go" areas may be added to the terrain file. The route planner then must know the format of this terrain representation in order to effectively and efficiently access the terrain information required to plan a particular route. A more sophisticated route planner can be implemented as the terrain becomes more difficult. For operations over desert terrain only, a simple route planner may suffice. Infantry operations in the Balkans, however, may require a more sophisticated route planning mechanism.

For the scenario generation process, the main consideration is that the mechanism is passed a start point and an end point and it returns a route. Because the architecture has been designed to be modular, more sophisticated route planners can be easily included. The steps to replace the current route planning mechanism with a more sophisticated algorithms are presented later in this chapter.

A simple route planner implemented for the purpose of prototyping uses the following technique. It is a relatively simple task for a human planner to look at a map and plan a route from point A to point B. The human planner will normally choose from several possible routes based on some criteria, such as ease of movement or avoidance of

enemy detection in planning a route for a military operation. Unfortunately, it is currently extremely difficult to give a computer this "bird's eye" view. If the problem is changed slightly, however, a simple search strategy can find a way around any number of obstacles. Assume a human planner is physically standing on the start point and looking out towards the objective point. The human planner could see any obstacles between the start point and the objective. He could then plan a route to avoid the obstacle. Another method is to assume that the human planner has partitioned the terrain into "go" and "no-go" areas. The "no-go" areas must be bypassed. He could then plan a route from the start point to the objective which bypasses all "no-go" areas.

A route planning algorithm can follow similar steps. If the terrain is divided into squares of some resolution with associated movement factors and elevation, the program uses an algorithm, such as Bresenham's Algorithm (9:70–72), to construct a line of sight between the two points. This method, when applied to this problem, determines which squares are between the start and objective points, analogous to the pixels on a screen. As the line of sight passes through a particular square, the grid square's mobility factor is examined. If that square is an obstacle, the computer can move (left or right) until the line of sight clears the obstacle. To find the edge of the obstacle, first the original line of march is continued until the back edge of the obstacle is found. Next the segment of the line within the obstacle boundaries is bisected. The next step is to begin plotting a line through the bisection point which is perpendicular to the original line of march. To ensure that the search finds the shortest point past the obstacle, one square to the right of the bisection point is generated and then one to the left. Then the algorithm generates a square two to the right and then one two to the left. As each square is generated, its mobility factor is checked. This process terminates as soon as a square is found that is not an obstacle. The problem then becomes planning a leg from the start to this new point past the obstacle and planning a leg from the point past the obstacle to the objective. Routes are represented as a list of points from the start point to the obstacle point, and then from this obstacle point to the objective. If another obstacle is encountered, a recursive call is the made and the process repeats until the objective is reached.

Bresenham's Algorithm can be modified to construct lines of more than a single pixel thick (9:72). This extendibility provides the ability to plan routes for multiple entities traveling abreast. Cunningham has developed techniques for calculating the penalty for routes where formations of entities would have to adjust their normal formations to travel (3:307-315). The source code for this route planner is Appendix C.

4.3.2 asgs.clp. This CLIPS program loads in the CLIPS rule bases of the Automated Scenario Generation System. Any other rule bases, such as new entities or new missions could be loaded automatically by adding the following statement to *asgs.clp*:

```
(load new_rules.clp)
```

This statement could be inserted between the last statement and the last parenthesis.

4.3.3 entity.clp. This CLIPS program defines the classes and subclasses for the objects for the Automated Scenario Generation System. This program is loaded automatically by the *asgs.clp* program. New objects can be added to this rule base in the following manner:

```
(defclass NEW_OBJECT
  (slot ATTRIBUTE 1)
  (slot ATTRIBUTE 2)
)
```

4.3.4 mission_definitions.clp. This CLIPS program defines the operations which a particular class of entities can perform. Each operation is defined as a CLIPS *message-handler* in the following manner:

```
(defmessage-handler NEW-OPERATION
    (Some function)
    (Some function)
)
```

4.3.5 driver.clp. This CLIPS routine provides the overall driver functions for the Automated Scenario Generation System. The *driver.clp* performs the following functions:

- Reads mission file entries
- Parses mission strings
- Instantiates player entities
- Assigns entities their respective missions
- Accepts completed mission plans from entities
- Formats completed mission plans to match target system
- Writes results to target system scenario file

4.3.6 scenhead.fil. This file contains information that is both common to the BATTLESIM model and specific to testing partitioning strategies for BATTLESIM. This information establishes the version number of BATTLESIM, the geographic area, battlefield sectors and the number of entities in the simulation. Because the primary purpose of the BATTLESIM model is research in improved parallel simulations and algorithms, the data elements specific to partitioning strategies are placed this file. This separation allows the system designer to easily and rapidly adjust partitioning strategies without effecting the original scenario file which reduces introduction of new errors into the system. The mapping of this information to the scenario file is presented later in section 4.3.9.

4.3.7 mission.fil. The mission file, *mission.fil*, provides a succinct statement of the task to be accomplished, entity responsible for the mission, entity description and key route points that are to be traversed during the operation. The format of the mission file is mission number, entity id, entity class, entity nomenclature, mission, execution time and mission start point and objective location. An example mission file is as follows:

```
1 A22 TANK M1A1 ROUTE-RECON 091200z_August_1993 08791234 05734321
```

4.3.8 BATTLESIM Scenario File. The BATTLESIM scenario file is an ASCII file. The scenario file can be divided into two sections: battlefield information and object information. Each line with a leading asterisk "*" is a comment line and ignored during execution by BATTLESIM (2:56). The next sections describe the non-comment lines found in the example BATTLESIM scenario file in Figure 9.

```
* version number
V4.0
* terrain data filename
terrain.10
* terrain min coordinates (x, y, z)
0.1 0.1 0.1
* terrain max coordinates (x, y, z)
0074.0 9999.0 1000.0
* number of sectors (must be < 64)
8
* sector min/max boundaries (x, y, z values in order
                            from 1st to last sectors)
0.1 59000. 0.1 29250. 118000. 1000.
29250. 59000. 0.1 58500. 118000. 1000.
58500. 59000. 0.1 87750. 118000. 1000.
87750. 59000. 0.1 117000. 118000. 1000.
0.1 0.1 0.1 29250. 59000. 1000.
29250. 0.1 0.1 58500. 59000. 1000.
58500. 0.1 0.1 87750. 59000. 1000.
87750. 0.1 0.1 117000. 59000. 1000.
* number of icon records
5
1 f18
2 mig1
3 missile
4 tank
5 truck
* object type through max climb
1 1 1 0 1 1 1 1000 0 0 0 0 0 1 1 1 1 1 1
* number of route points
4
* route coordinates x, y, z (start to finish order)
0.1 0.1 1000.0
2.0 3.0 1000.0
40.0 55.0 1000.0
100.0 100.0 1000.0
* number of sensors
1
1 5850 1
* number of armaments
0
* armament description (if above > 0)
* number of targets
1
* target descriptions (if above > 0)
1 0 0 0
* number of defensive systems
0
* defensive system descriptions (if above > 0)
* END OF OBJECT
```

Figure 9. BATTLESIM Scenario file

4.3.9 Battlefield Information. Figure 10 shows the first section of the BATTLESIM scenario file. This section describes some general battlefield and set up information required by BATTLESIM. This information is found in all scenario files and is common to all objects. This information is unique to BATTLESIM and is stored in a file called *scenhead.fil.* The information is stored in the format displayed in Figure 10 and is copied directly from this file to the scenario file. This design allows researchers to easily change information that is unique to parallel processing without changing information in the mission file which is unique to the scenario generation process.

```
* version number
V4.0
* terrain data filename
terrain.10
* terrain min coordinates (x, y, z)
0.1 0.1 0.1
* terrain max coordinates (x, y, z)
0074.0 9999.0 1000.0
* number of sectors (must be < 64)
8
* sector min/max boundaries (x, y, z values in order
                            from 1st to last sectors)
0.1 59000. 0.1 29250. 118000. 1000.
29250. 59000. 0.1 58500. 118000. 1000.
58500. 59000. 0.1 87750. 118000. 1000.
87750. 59000. 0.1 117000. 118000. 1000.
0.1 0.1 0.1 29250. 59000. 1000.
29250. 0.1 0.1 58500. 59000. 1000.
58500. 0.1 0.1 87750. 59000. 1000.
87750. 0.1 0.1 117000. 59000. 1000.
* number of icon records
5
1 f18
2 mig1
3 missile
4 tank
5 truck
```

Figure 10. BATTLESIM Scenario File Battlefield information

Because the primary purpose of the BATTLESIM model is research in improved parallel simulations and algorithms, the data elements specific to partitioning strategies

```
* object type thru max climb
1 1 1 0 1 1 1 1000 0 0 0 0 0 0 1 1 1 1 1 1
* number of route points
4
* route coordinates x, y, z (start to finish order)
0.1 0.1 1000.0
2.0 3.0 1000.0
40.0 55.0 1000.0
100.0 100.0 1000.0
* number of sensors
1
1 5850 1
* number of armaments
0
* armament description (if above > 0)
* number of targets
1
* target descriptions (if above > 0)
1 0 0 0
* number of defensive systems
0
* defensive system descriptions (if above > 0)
* END OF OBJECT
```

Figure 11. BATTLESIM Object Information

are placed in a separate file called *SCENHEAD.FIL*. These data elements include the version of BATTLESIM, the terrain file which is being used, the maximum and minimum coordinates of the battlefield, the number of sectors and the boundaries of each of those sectors. Because BATTLESIM does not consider terrain during execution, coordination between the BATTLESIM maximum and minimum coordinates and the prototype system terrain representation is done manually.

The separation of this header information from the rest of the scenario file allows researchers to easily and rapidly adjust partitioning strategies without effecting the original scenario file which reduces introduction of new errors into the system.

4.3.10 Object Information. Figure 11 shows an example of the information normally found in the object information portion of a BATTLESIM scenario file.

The "* object type through max climb" entry is a string of values which describes the critical values of the object. The fields of the "object type through max climb" are defined below:

- object type - identifies icon which is associated with a particular object.
- object id - unique integer which is assigned to the objects in a simulation session.
- object loyalty - integer which identifies which objects are friendly and which are enemy. All objects with the same number are friendly and will not attack each other, while those with different numbers are enemies and may attack each other.
- current time - current simulation time of the object.
- object fuel status - amount of fuel that a object has left. It is not used in this version of BATTLESIM.
- object condition - damage condition of an object. It is not used in this version of BATTLESIM.
- object vulnerability - amount of destructive force required to destroy the object. This attribute is currently not used in BATTLESIM.
- object x,y, and z velocity - an object's x,y and z velocity vectors.
- object yaw, pitch, and roll rate - an object's orientation. These attributes are not used in the current version of BATTLESIM.
- object experience and threat knowledge - experience and threat knowledge of the operator of an object. These values are not used in the current version of BATTLESIM.
- object performance - last four fields of the "object type through max climb string" describe some performance characteristics of the object. These four fields are:
  - minimum turning radius
  - max speed
  - average fuel consumption
  - max climb rate

This information consists of default values based on the class of the entity. By querying the object through use of the CLIPS *send* command, these data values can be obtained from the entity. For example, the CLIPS command to retrieve a value from the *nomenclature* slot of a particular object, A22 is:

```
(send [A22] get-nomenclature)
```

Although the object specific information can be obtained directly from an object, the route information cannot be. This data is not directly read from any source. Rather

the source of this information is the results of the intelligent mission planning. After reading the mission file, instantiating the proper objects and passing them their missions to plan, mission planner waits until the object passes the completed mission plan back to the mission planner. At this point, the mission planner formats the returned mission plan and writes the results to a scenario file. In the case of BATTLESIM, a list of route points (x, y, z coordinates) are created for each individual entity.

The source of the information in the final portion of the scenario file is the default values from the particular object class. This information represents the weapon systems and defensive capabilities of the object. These values can be obtained by querying the object through by using the CLIPS *send* command.

- Sensor Description - the number, type, and range of the sensors.
- Armaments Description - the number and type of weapon systems which an object possesses.
- Number of Targets - the number of targets which are included in an entity's target list.
- Target Descriptions - the type and location of targets.
- Number of Defensive Systems - the number of defensive systems which an entity has.
- Defensive System Descriptions - entity's defensive system, such as chaff or electronic counter-measures.
- End of Object - flag which marks the end of an object's description.

4.4 Intermediate Format to Scenario File Mapping

Section 4.3.8 discussed the format required for the BATTLESIM scenario file and Section 3.5.3 explained the mapping of the operation order to the intermediate format. This section concentrates on the mapping from the intermediate format to the BATTLESIM scenario file.

4.4.1 Scenario File Header Information. For the purpose of this research, the information prior to the route information is not altered. This information is stored in the *scenhead.fil* in the format illustrated in Figure 10. These data elements are copied directly from *scenhead.fil* into the scenario file. This design provides easy and rapid adjustments

to partitioning strategies without effecting the original scenario file. This method reduces the likelihood of the introduction of new errors into the system.

4.4.2 Mission File to Scenario File. Each record in mission file consists of eight to eleven fields in the following format:

- mission number
- entity id
- entity class
- entity nomenclature
- mission
- date time group of mission
- start point
- objective
- optional zero to three checkpoints

Each record in the mission file is read, parsed and processed. This processing consists of instantiating an object called "entity id" of class "entity class". After the object is instantiated, key information is obtained from the object using the CLIPS *send* command. This information obtained directly for the object maps directly to to the following fields in the scenario file:

- object type through max climb
- number of sensors
- number of armaments
- armament description
- number of targets
- target descriptions
- number of defensive systems
- defensive systems descriptions

The route information of the scenario file results from the intelligent mission planning of each object. First, the instantiated entity plans a route based on the assigned mission, start point, objective point and any checkpoints. The completed route is returned to *driver.clp* which counts the points and writes the number of route points and the individual points to the scenario file.

4.4.3 Complete Scenario File. Appendix H shows a sample mission file and Appendix I is the resulting scenario file that was generated.

4.5 Testing

To validate and demonstrate the tractability of generating a scenario file from a mission file, scenarios were generated from mission files under a variety of starting conditions. These starting conditions included single versus multiple entities, varying numbers of obstacles and varying numbers of intra-route checkpoints.

4.5.1 Route-Recon Mission Testing. Figure 12 represents a simple one entity test. The entity is to conduct a route recon from its current location in Assembly Area Blue to Objective Dog. For example, a platoon operations order might contain in paragraph 3, *Execution*, the following subordinate unit mission for a tank crew to conduct a route recon:

```
A22 will conduct a route-recon from AA Blue (vic. WP00010001) to Objective
Dog (vic. 00030070) commencing at 230600z_Aug_93.
```

This mission would be processed into the following intermediate format representation:

```
1 A22 tank M1A1 route-recon 230600z_Aug_93 00010001 00030070
```

Using this scenario file as input to the mission planner results in the creation of a scenario file for a single entity with route points generated between 00010001 and 00030070, avoiding any obstacles which may lie in the way. The route selected depends on the nature of the terrain and the route planning algorithm selected.

Checkpoints are often used in operation orders to control the movements of units during an operation. Because of this fact, an added feature allows the specification of checkpoints. These checkpoints are introduced in the following manner:

```
1 A22 tank M1A1 route-recon 060023zAug93 00010001 00030029 00020010 00030005
```

The route checkpoints are added to the record after the objective point. For the purpose of demonstration, the prototype system provides for up to three checkpoints. The order of

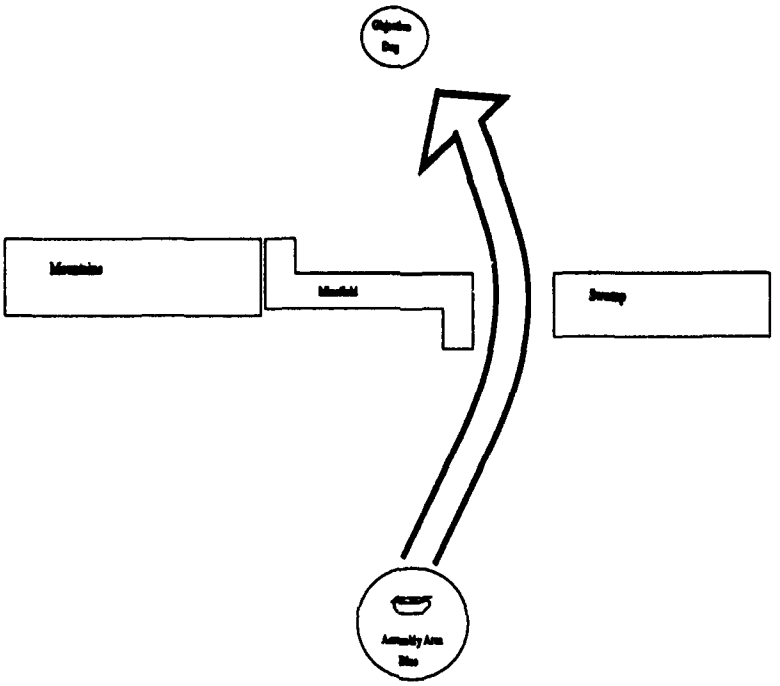

Figure 12. Single Tank Conducting a Route Recon

traversal for a route-recon mission is:

$$StartPoint \rightarrow Checkpoint1$$

$$Checkpoint1 \rightarrow Checkpoint2$$

$$Checkpoint2 \rightarrow Checkpoint3$$

$$Checkpoint3 \rightarrow ObjectivePoint$$

Figure 13 shows graphically one way a checkpoint is specified and Figure 14 shows the results of scenario generation session for planning a route from 00150001 to 00030029 including a checkpoint located at 00200010. This checkpoint is a mandatory traversal point and must be included in the final route. The mission statement from which such a mission file entry would be derived could be:

```
A22 will conduct a route-recon from present location Assembly Area Blue
(vic. WP00150001) through Checkpoint1 (vic. 00200010) to Objective Dog
(vic. 00030029) commencing at 230600z_Aug_93.
```

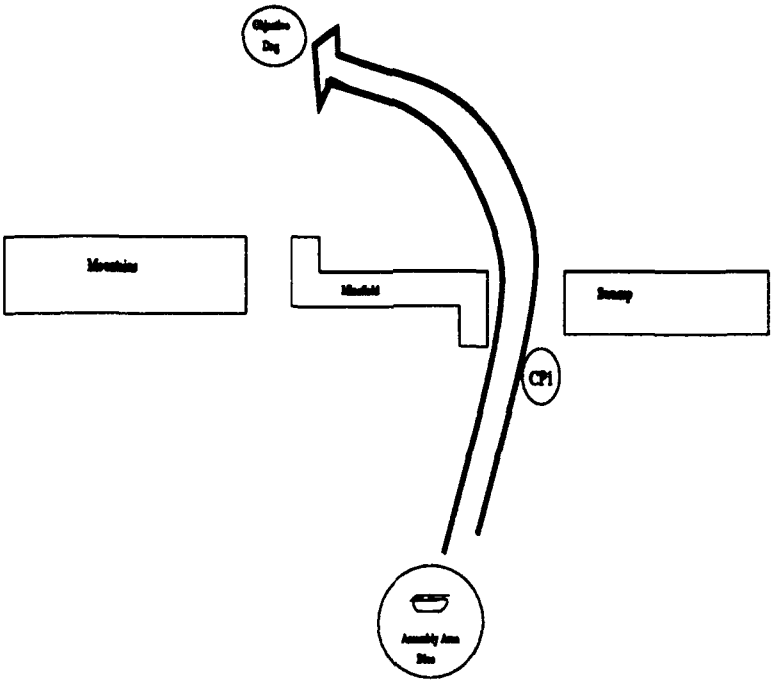

Figure 13. Single Tank Conducting a Route Recon through CP1

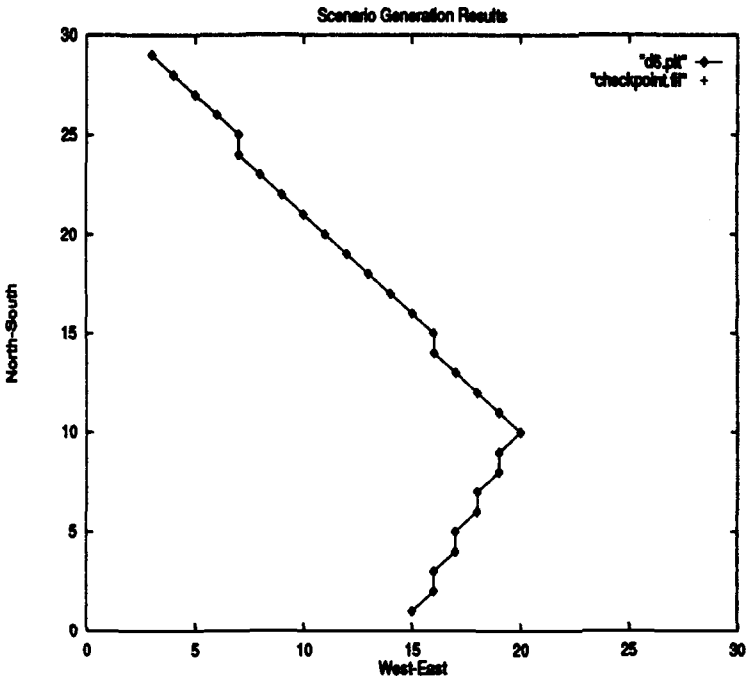

Figure 14. Mission Planning with Checkpoints included

Scenario files were generated for routes involving multiple obstacles as well. The algorithm used for this prototype is such that the route planning routine will continue to make recursive calls until the objective point is reached. For example, if the route planner finds an obstacle between the start and the objective, it will generate a point at the edge of the obstacle. The system then runs the route planning algorithm from start to the edge of the obstacle and then from tangent point to the objective. If another obstacle is found in those two segments the system makes a recursive call to itself and continues until the objective is found.

Figure 15 displays the results of planning a route from point 00130003 to point 00070023 with four obstacles. Figure 16 is the same mission with the start and end points reversed. The difference in the basic route between these two plots is caused by the the ordering of the planning mechanism. The route planning algorithm checks the right square before the left and exits as soon as it finds a grid square which is not an obstacle. Consequently, if the original line of march is North to South and bisects an obstacle, the route will always try to go around to the West first. Conversely, if the original line of march is South to North and bisects an obstacle, the route planning mechanism will try to go to the East first. The mission planning system successfully generated routes avoiding multiple obstacles as presented in Figures 15 and 16.

4.5.2 Bomb-Target Mission Testing. The bomb-target mission is unique to the *plane* class of objects. This mission consists of planning a route to a target and back to the start point. Checkpoints can be included, but are used in a slightly different manner than the route-recon mission. Only two checkpoints will be used during the bomb target mission and the rest will be ignored. The first checkpoint is used in planning a route to the objective and the second is used to plan the route from the objective.

The other key difference in planning this mission is that the entities will have target lists generated and entered into the corresponding entity slots. The bomb-target mission makes use of the same route planning mechanism as the route-recon mission.

Figure 17 shows graphically a simple bomb-target mission. The plane is to start from an airfield in Assembly Area Blue, fly through Checkpoint1 avoiding enemy radar

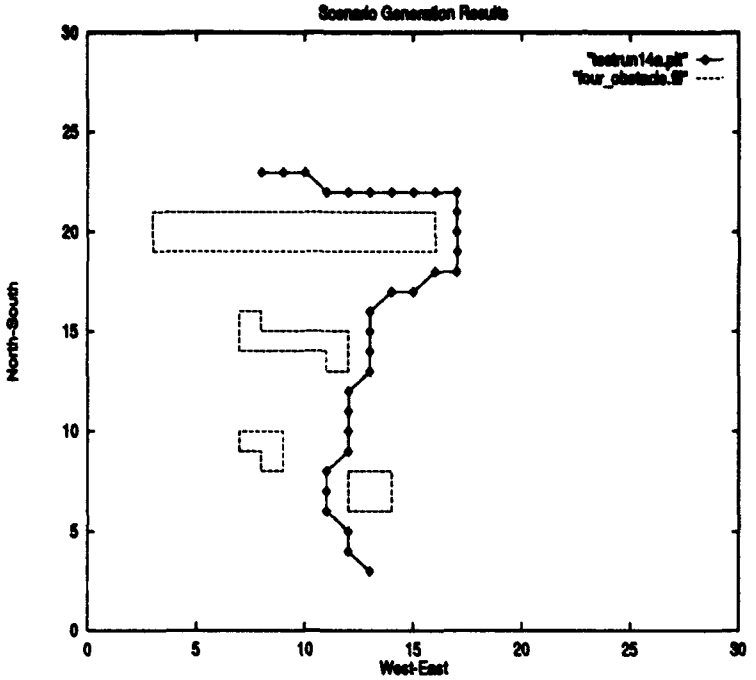

Figure 15. Mission Planning with Four Obstacles, moving northward

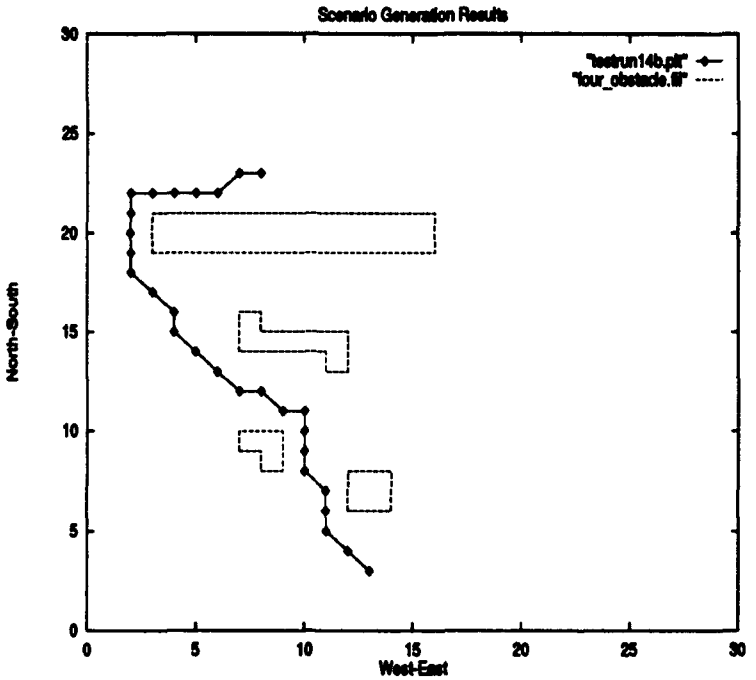

Figure 16. Mission Planning with Four Obstacles, moving southward

and mountain ranges to bomb a target in Objective Dog. The plane is to then return by a different route through Checkpoint 2.

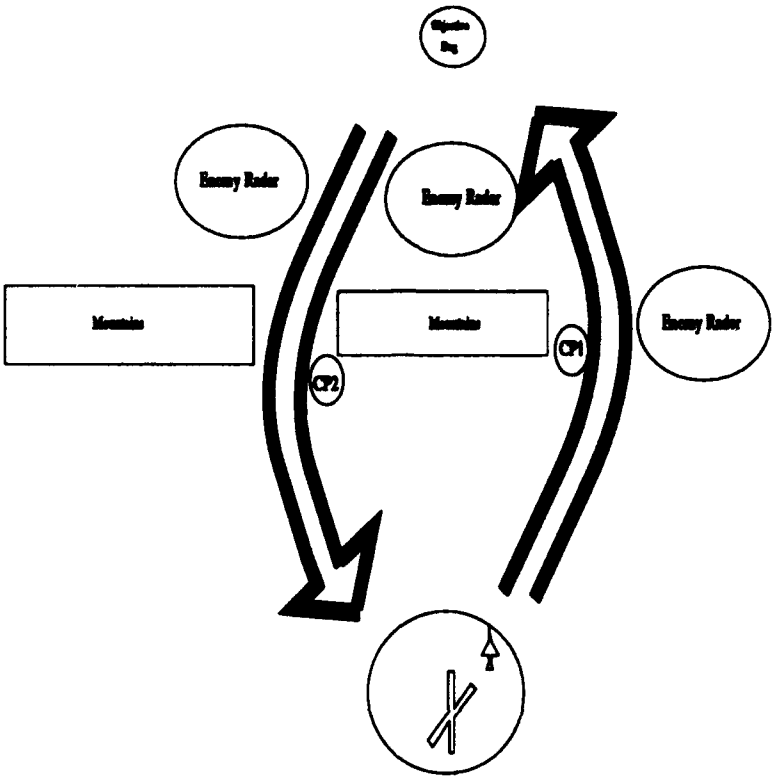

Figure 17. Simple Bomb Target Mission

Paragraph 3, *Execution* could have the following subordinate unit instructions:

```
Plane T123 will depart the airfield in Assembly Area Blue at 232300z_AUG_93 to
conduct a bombing mission on a communications facility in Objective Dog
vic. 00030070. Movement control measures of Checkpoint 1 on initial leg
and Checkpoint 2 on return leg are in effect.
```

These statements would then be processed into the following entry in the mission file:

```
1 T123 plane F18 bomb-target 232300z_AUG_93 00010001 00030070 00010010 00040060
```

4.5.3 Combined Arms Scenarios. In order to validate the idea of generating scenarios involving multiple entities, more complex mission files were used. This time the

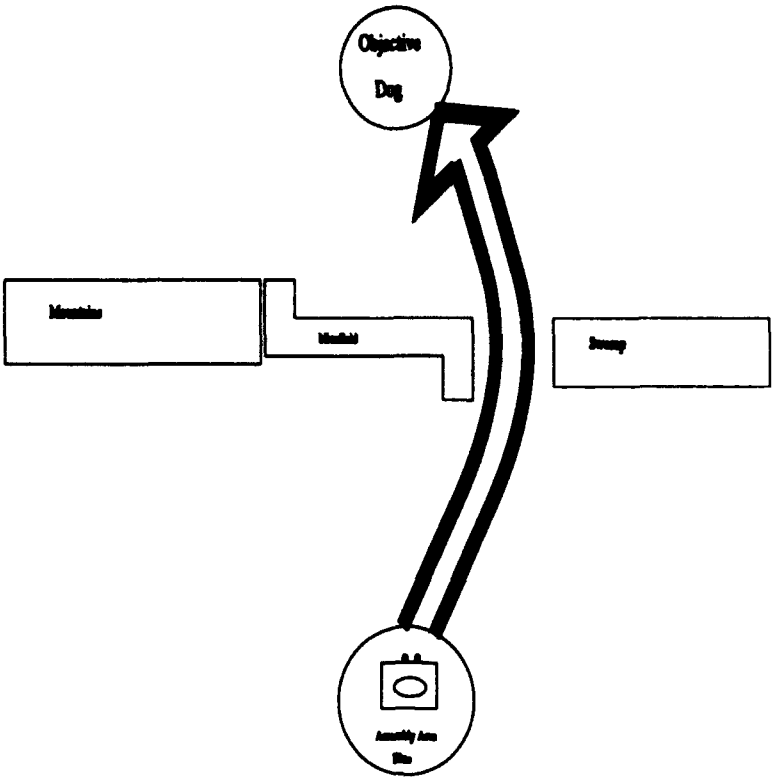

Figure 18. Tank Section Example

mission file consisted of multiple line entries, which represents multiple entities performing their respective missions. An example mission statement for two entities could be:

```
1st section of 1st platoon of A Company will conduct a route recon from
current battle positions (vic. 00010001 and 00130003) to new battle positions
(vic. 00190020 and 00040017) commencing at 230600z_AUG_93 and
230800z_AUG_93.
```

Figure 18 show the graphic representation of this mission. Since normal tank sections consist of two tanks, the automated scenario generation system would create the following mission file:

```
1 a11 tank M1A1 route-recon 060023z_AUG_93 00010001 00190020
2 a22 tank M1A1 route-recon 070024z_AUG_93 00130003 00040017
```

This test resulted in the creation of a scenario file for multiple entities performing their particular missions. The planning mechanism still functions in the same manner, however. Each line of the mission file is read and processed one at time. There is currently

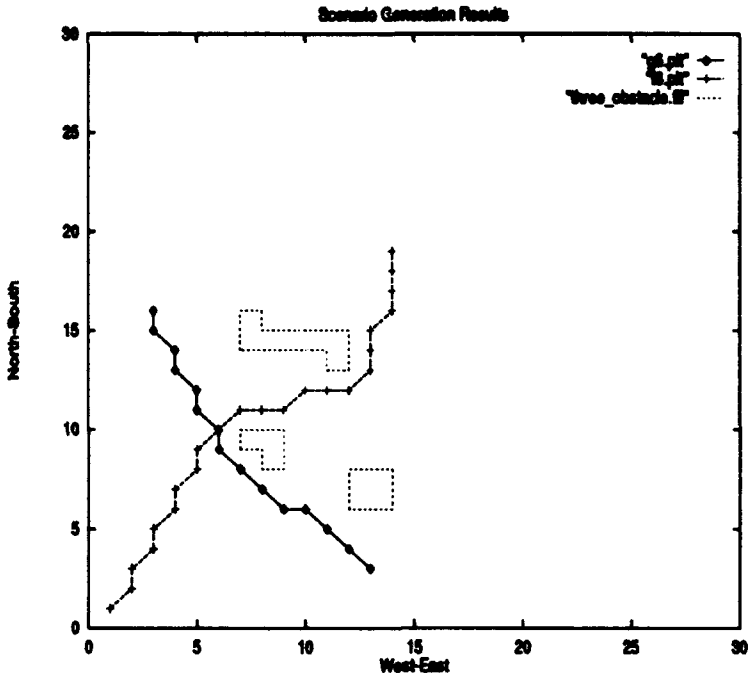

Figure 19. Mission Planning with multiple entities and obstacles

no provision for route de-confliction in the planning portion of this system. Route de-confliction could be added by replacing the route planner with a more sophisticated route planner. The procedure for modifying the route planning mechanism is described in Section 4.7.2.

Testing multiple entities was conducted in a manner similar to the single entity testing, starting with simple routes with no obstacles or checkpoints, and progressing to more difficult scenarios. Figure 19 displays the results of a mission file with two entities and three obstacles. The next logical step was to test the Automated Scenario Generation System's ability to generate scenario files for combined arms missions. Combined arms missions consist of multiple entities of different classes with different missions. Figure 20 illustrates a combat situation where a plane object conducts a bombing mission while two tank objects move toward the same objective. This mission could be assigned to the participants in the operation in the following manner:

A22 will move to occupy Objective Dog from its current location in Assembly Area Blue. B23 will move to occupy Objective Dog from its current location in Assembly Area Red. T123 will conduct a bombing mission from its current

location at the airfield in Assembly Area Green to Objective Dog to neutralize enemy forces on Objective Dog.

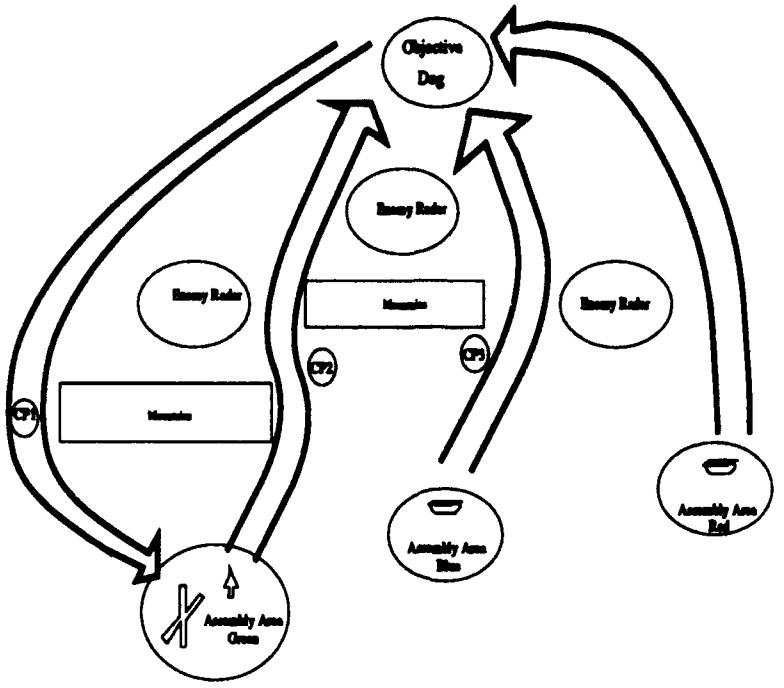

Figure 20. Example Combined Arms Mission

The mission file entries generated from this portion of an operation order would be:

```
1 A22 tank M1A1 route-recon 232300z_AUG_93 00500010 01000100
2 A23 tank M1A1 route-recon 232300z_AUG_93 00700020 01000100
3 T123 plane F18 bomb-target 232300z_AUG_93 00100010 01000100 00500080 00010080
```

The automated scenario generation system then creates a scenario file with these three entities and their respective routes included.

4.5.4 Linear Obstacles. Linear obstacles are defined in this paper as terrain features such as rivers, canyons and mountains. The common relationship of these types of features is that they are usually much longer than wide, are generally oriented perpendicular to an object's line of march and are of a nature that constitutes "no go" terrain. Figure 21 shows the results of a route-recon mission successfully planned from the point 00070003 to the point 00210022. The obstacle marked by the lines represents a river. A bridge, located at 00280015, crosses the river. The line with box points represents the route points generated by the route planner.

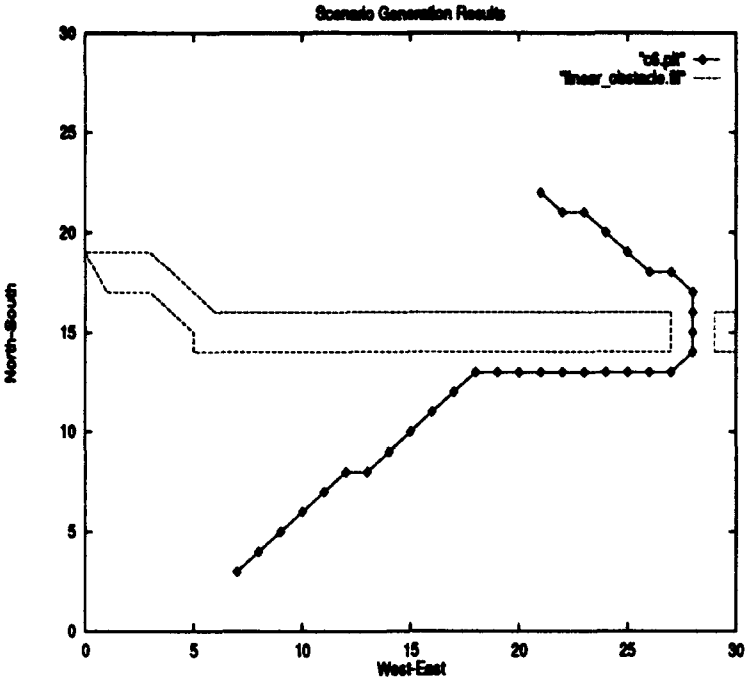

Figure 21. Successful Linear Obstacle Example

These obstacles cause the route planner significant problems. Figure 22 displays the same river obstacle, but with different start and end points. The route appears to go to the river and then veer off snarply to end at the point 00000000. Actually the route planner continues trying to generate grid squares and never progresses to the goal. This leads to an out of memory condition which then aborts the system.

More sophisticated route planning algorithms can be substituted in place of this simple route planner to handle every possible case of linear obstacles. The method for substituting more sophisticated modules is presented later in this chapter.

4.6 Performance

Two measures of performance are presented. First, the BATTLESIM compatibility issues are addressed. Part of this examination is the results of testing the BATTLESIM generated graphics files in the VISIT system. Second, the speed of the system in generating small to large scenarios is examined.

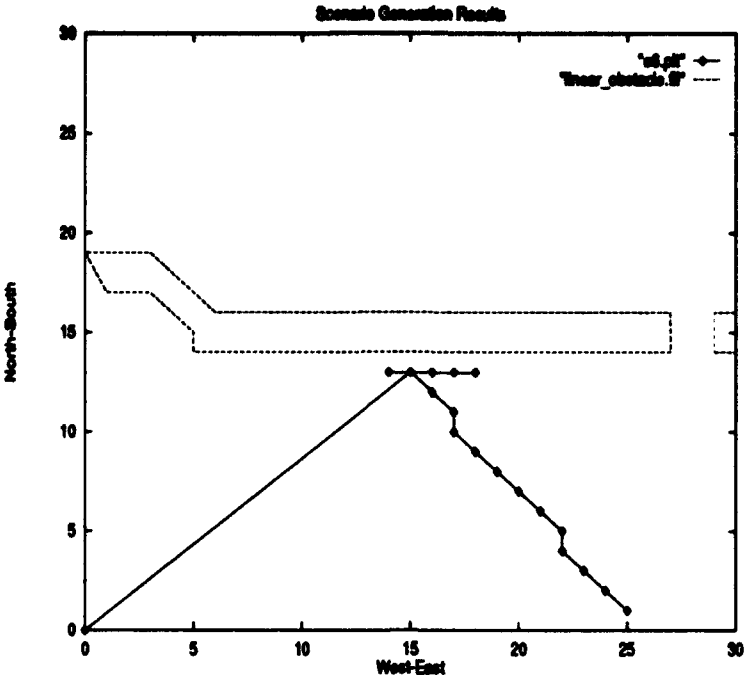

Figure 22. Unsuccessful Linear Obstacle Example

4.6.1 BATTLESIM and VISIT Compatibility. The scenario files generated as the result of the tests described in the previous sections were processed through BATTLESIM. Single and multiple entity mission files were created and processed. The resulting scenario files were then loaded into BATTLESIM and executed. The files ran in BATTLESIM without problems. The VISIT system is a graphical display system designed by DeRouchey (7) to view the results of BATTLESIM simulation sessions. The graphical output file of each BATTLESIM run was executed in the VISIT system. The proper icons were displayed and performed the proper actions. No problems were noted in performance for any size scenario file tested.

4.6.2 Speed. Test scenario files were generated for 1 to 25 entities with each entity planning a mission over a route varying from 3 kilometers to 20 kilometers. The minimum time to generate a scenario for a single entity mission over 3 kilometers was less than 4 seconds. The maximum time to generate a scenario was for 20 entities over routes of 20 kilometers each. This scenario creation time was 10 minutes.

Two general observations are noted in the relationships between number of entities, length of route and processing time. The first observation is that as the number of entities in the mission file increases, the time that the prototype system requires to generate the scenario file likewise increases. This increase in time can be attributed to the fact that the prototype system executes the planning process for every entity in the mission file. The increase in scenario generation time seemed to be linear, if the length and difficulty of the routes for the additional entities were held constant. For example, the time to generate the same route for ten entities tended to be approximately ten times longer than for one entity.

The other general observation noted is that longer routes require longer times to plan than shorter routes. This increase can be attributed to the fact that each grid square along the route is generated. Longer routes consist of more grid squares than shorter routes and therefore, require more time to plan. Also, the complexity of each route tended to increase the planning time for a particular route. Generally, the more obstacles encountered, the longer the system required to plan a route. Neither of these increases seemed to be linear.

4.7 Modularity

One key point of the object oriented design paradigm is the capability to replace modules without substantial re-engineering of the entire system. This system architecture has been designed with that replaceable concept. This versatility can be demonstrated in the flexibility of the prototype in adding new features.

4.7.1 New Entities. New entities can be easily added to entity.clp. The only requirement is that if the entity is not a member of an existing class, then a message-handler function would also have to be added to *mission_definitions.clp.* The message-handler must be added to the *mission_definitions.clp* in order to provide a mechanism to assign values to and read data from a particular entity. CLIPS message-handlers are specific to a class. No other changes would have to be made to any other portions of the system.

4.7.2 New Route Planner. Another example of the modularity of this design is the route planner. A new route planning mechanism can be added two different ways. First, the existing CLIPS extension routines in the file called *r_plan.c* could be removed and replaced with the new route planning algorithm. To minimize changes to the existing source code, the new route planning routine would need to conform to two standards. The name of the called routine should be the same as the original which is *route_planner*. Also, the new route planner should return a string of route points. The *driver.clp* routine accepts a string from *route_planner*, parses out the individual route points and writes these route points in floating point notation to a scenario file. Changes to this original design, such as returning individual route points would require changes in other parts of the system. Second, a route planner could be written in CLIPS and loaded at run time. Again the only requirement would be that the new route planner return a string of route points to the Automated Scenario Generation System.

4.7.3 New Missions. Because the missions are written as CLIPS rules, adding new missions to the *missions_definitions.clp* file is a straightforward process. New missions require a new message-handler to be defined in the *mission_definitions.clp* file. Message-handlers are used by CLIPS to access data fields in an object. These message-handlers are the only methods for writing data to and reading data from an object. The naming convention used in this system is to name the message-handler the same as the new mission. This convention provides a straightforward mapping from the operation order to the mission file and from the mission file to the automated scenario generation system.

Associated rules to conduct the mission are also added to this file. Each mission requires the addition of rules which handle the conditions specific to the new mission. To add a *movement to contact* mission, for example, might require defining rules to handle actions if attacked by enemy artillery or if hit by an ambush.

The difficulty lies in making sure that the new rules do not cause an error in existing missions. This problem is one that is associated with rule base systems in general and not a problem specific to the implementation of this prototype.

4.7.4 New Terrain Representations. Currently, terrain is stored as an ASCII text file which is referenced by special access functions like *getgrid*. Future implementations could include terrain representations stored in relational database systems. This could provide access to a wide variety of terrain databases from the Corps of Engineers and the Defense Mapping Agency. To minimize changes to the existing source code, the new terrain module should return a string which represents an eight digit grid point.

4.7.5 New Target Simulation System. The target simulation system for the prototype developed in this research is the AFIT BATTLESIM system. The architecture, however, provides the flexibility to modify the system to work for a variety of other systems. Converting the current prototype to a new target system is a three step process. The first step is to examine the format of the scenario file of the new system. This examination determines the source of the information for each required entry. Scenario information that is administrative in nature or common to all objects can be placed in a scenario header file in the same manner that SCENHEAD.FIL is created for the current target system. The second step is to add any new programs that might be necessary for determining additional information required by the new target system which is not provided for in the current prototype system. The *asgs.clp* program would be modified to load these additional programs in as required. The third step is to replace *driver.clp* with a CLIPS program which reads the mission file, instantiates the proper objects, assigns them missions, receives the completed mission plans from the objects and then writes the completed mission plans scenario file in the format of the new target system.

Depending on the new target system, changes may have to be made to *entity.clp* to add or delete objects supported by the new system or to *mission_definitions.clp* to add or delete missions supported by the new system. Modifications to these programs are described in previous sections.

4.8 Conclusion

This chapter presented the implementation of a prototype automated scenario generation system based on the design proposed in Chapter 3. This chapter also detailed

the results of testing the prototype system under a wide range of possible situations. The chapter then demonstrated the modularity of the architecture by demonstrating how the working prototype could be easily and rapidly modified to include additional objects, more sophisticated route planning mechanisms, more sophisticated terrain models, and additional missions. The final portion of this chapter described the portability of the prototype to support other target simulation systems.

V. Results, Conclusions and Recommendations

5.1 Introduction

This chapter summarizes the work accomplished during this research effort. Section 5.1 discusses the results of the research. Section 5.2 compares these results to the original research objectives. Section 5.3 gives limitations of the prototype system. Section 5.4 recommends topics for follow-on research.

5.2 Results

The objective of this effort was to design a flexible Architecture for Automated Scenario Generation Systems which will support reading text orders and creating scenario files with minimal user interaction, and to develop a prototype system to provide proof of concept for possible follow-on research in this area. Both of these goals were successfully accomplished.

5.2.1 Design. The Architecture for an Automated Scenario Generation System proposed in this thesis provides the critical capabilities to fully automate the scenario generation process. The design starts with the common military operations order as input, processes it, plans the missions found in the operations order, and creates a scenario file in the proper format for the target simulation system.

The object oriented design of this architecture allows for "upgrades" of each key portion of the system. For example, as improved natural language processing algorithms are developed, these improvements can be incrementally substituted into the system. These substitutions should require few, if any, changes to the other portions of the system.

5.2.2 Prototype Implementation. The prototype system presented in this thesis generated scenario files from an intermediate format. These scenario files are fully BATTLESIM compatible and were extensively tested using the BATTLESIM model.

The time that the prototype required to produce scenario files ranged from seconds in the case of small scenarios to over 10 minutes for larger scale scenarios.

The prototype system reads the intermediate format file referred to as the mission file, generates the proper entity type and successfully plans missions for any number of entities.

5.3 Limitations

The prototype has a number of limitations. Several areas are examined in the next three sections.

5.3.1 Route Planner. The prototype provides a rudimentary route planning mechanism with a number of shortcomings. The first shortcoming is that algorithm finds a path from the start point to the objective, but does not guarantee the shortest or optimal path. This problem is caused by the route planning mechanism terminating the search process immediately after finding a path to the objective.

A second limitation is that the prototype has a limit to the number of route points which can be handled for a given mission. Routes over thirty kilometers have caused intermittent segmentation fault errors to occur during execution. Routes of less than 30 kilometers seem to pose no difficulties for any number of entities.

A third limitation of the route planner is the type of obstacles which can be avoided by the route planner. One assumption of this research effort was that only convex obstacles would be considered. Minimal testing was conducted including concave obstacles. The route planner's success or failure depended heavily on the obstacle's orientation to the entity's line of march. Additionally, linear obstacles often caused the route planner to fail, as explained in Section 4.5.4

5.3.2 Checkpoints. The prototype allows 0, 2, or 3 checkpoints to be included in the mission file. The number of valid checkpoints and the processing of those checkpoints depends on the type of mission. One-way missions, such as recon-route, allow 0-3 checkpoints to be included in the mission file. Two-way missions, such as bomb-target, allow either 0 or 2 checkpoints only. This limitation is caused by the design of *driver.clp* module. This module reads in the mission file, parses out the key components, instantiates

the entity, and passes the mission to the entity for planning. The *driver.clp* module is set to only look for a maximum of three checkpoints. To modify the prototype to allow for more checkpoints, *driver.clp* would be changed to look for more checkpoints. The *mission_definitions.clp* module would also need additional rules added to handle cases of more than three checkpoints.

5.3.3 Text Processor. An assumption of this research effort was that sufficiently sophisticated natural language processing systems could be developed to transform a textual operation order into an intermediate format. A simple text processor was designed and implemented in the *awk* computer language. The text processor found at Appendix L processed the simple operation order found at Appendix M. The mission file which resulted from this process is Appendix N.

The prototype text processor uses simple pattern matching and depends on a very structured operation order as input.

5.4 Future Recommendations

5.4.1 Parallel Implementations. This system could be improved by porting it to a parallel system. Each entity could independently plan its individual mission on a separate processor. The time to generate a large scale scenario could be drastically reduced since multiple concurrent planning processes could occur simultaneously.

5.4.2 Other Terrain Representations. Current methods for terrain representation are poorly suited for this type of problem. Implementing representations which provide easier access to information will improve this system. As newer algorithms and storage methods are developed, improved scenario generation systems can be constructed.

Interfaces to existing terrain files in relational databases could provide significant benefits in the area of mission planning. The Automated Scenario Generation System could query the database through SQL calls. If the database returns the data in the format understood by the Automated Scenario Generation system, minimal modifications would be necessary to the current model.

5.4.3 Improved Mission Planning. This system was implemented in CLIPS version 5.0. CLIPS is useful tool to develop rule-based systems. Further research into other inference engines may discover other more powerful and versatile development tools.

Additionally, other missions, such as "movement to contact" and "retreat", could be implemented and tested. The steps to implement this modification are described in Chapter 4. Any additional rules added to the *mission_definitions* would have to be checked against the existing rules to ensure that no conflicts exist. This problem is a consideration of modifying most rule-based systems.

5.4.4 Natural Language Processing. High quality Natural Language Processing (NLP) systems could be integrated, allowing the system to perform the entire scenario generation process. More sophisticated natural language processing systems could process operation orders in languages other than English, verbal orders and briefings. The NLP system would process these inputs into an intermediate format which could then be processed by into a simulation specific scenario file.

5.5 Conclusion

Based on currently available literature and the lack of sophisticated models, the area of automated scenario generation for combat simulations seems to have been an area of limited research. Recent work in simulation models, such as TRADOC's *Eagle* and *CASTFOREM*, has greatly improved the user interface. These improvements, while substantial, have not fully automated the process.

The thrust of this research effort was to determine the requirements for fully automating the scenario generation process, defining an architecture which fulfilled these requirements and then designing a working prototype to demonstrate that the proposed architecture was valid. Based on the performance of the prototype in generating large fully BATTLESIM compatible scenario files, this architecture seems to support the goal of fully automating the scenario generation process. The prototype is not perfect in that the natural language processing portion was not implemented, a simple route planning mechanism was used and mission planning by each entity is rudimentary, but the performance of the

prototype does suggest that the idea of accepting a military operation order as input, processing it through natural language processing into an intermediate format, passing that intermediate format to an intelligent agent for mission planning and then accepting the results back from the agent to record in a scenario file is possible and could significantly enhance the scenario generation process for many combat simulations.

Appendix A. Example BATTLESIM Scenario File

```
* version number
V4.0
* terrain data filename
terrain.fil
* terrain min coordinates (x, y, z)
0.1 0.1 0.1
* terrain max coordinates (x, y, z)
0074.0 9999.0 1000.0
* number of sectors (must be < 64)
8
* sector min/max boundaries (x, y, z values in order from 1st to last sectors)
Must be user mapped
* number of icon records
3
1 f16
2 truck
3 tank
*object type through max climb
1 1 1 0 1 1 1 1000 0 0 0 0 0 1 1 1 1 1 1
* number of route points
4
* route coordinates x, y, z (start to finish order)
0.1 0.1 1000.0 2.0 3.0 1000.0 40.0 55.0 1000.0 100.0 100.0 1000.0
* number of sensors
1 5850 1
* number of armaments
0
* armament description (if above > 0)
* number of targets
1
* target descriptions (if above > 0)
1 0 0 0
* number of defensive systems
0
* defensive system descriptions (if above > 0)
* END OF OBJECT
```

```
   /*******************************************************/
   /*      "C" Language Integrated Production System      */
   /*                                                     */
   /*                  A Product Of The                   */
   /*             Software Technology Branch              */
   /*             NASA - Johnson Space Center             */
   /*                                                     */
   /*             CLIPS Version 5.00  11/19/90            */
   /*                                                     */
   /*                     MAIN MODULE                     */
   /*******************************************************/
/*************************************************************/
/* Purpose:                                                  */
/*                                                           */
/* Principal Programmer(s):                                  */
/*      Gary D. Riley                                        */
/*                                                           */
/* Contributing Programmer(s):                               */
/*                                                           */
/* Revision History:                                         */
/*                                                           */
/*************************************************************/
#include <stdio.h>
#include <dos.h>
#include "setup.h"
#include "sysdep.h"
#include "commline.h"

#if ANSI_COMPILER
int main(int,char *[]);
VOID UserFunctions(void);
#else
int main();
VOID UserFunctions();
#endif
/*************************************************************/
/* MAIN: Start execution of CLIPS.  This function must be    */
/*   redefined in order to embed CLIPS within another program. */
/*   Example of redefined main:                              */
/*     main()                                                */
/*       {                                                   */
/*        init_clips();                                      */
/*            .                                              */
/*            .                                              */
/*            .                                              */
/*        process_data();                                    */
/*        RunCLIPS(-1);                                      */
/*        evaluate_data();                                   */
/*            .                                              */
/*            .                                              */
/*            .                                              */
/*        final_results();                                   */
/*       }                                                   */
/*************************************************************/
int main(argc,argv)
  int argc;
  char *argv[] ;
  {
```

```
   InitializeCLIPS();
   RerouteStdin(argc,argv);
   CommandLoop();
   return(-1);
  }

/*************************************************************/
/* UserFunctions:  The function which informs CLIPS of any   */
/*   user defined functions.  In the default case, there are */
/*   no user defined functions.  To define functions, either */
/*   this function must be replaced by a function with the   */
/*   same name within this file, or this function can be     */
/*   deleted from this file and included in another file.    */
/*   User defined functions may be included in this file or  */
/*   other files.                                            */
/*   Example of redefined UserFunctions:                     */
/*     UserFunctions()                                       */
/*       {                                                   */
/*        DefineFunction("fun1",'i',fun1,"fun1");            */
/*        DefineFunction("other",'f',other,"other");         */
/*       }                                                   */
/*************************************************************/
void UserFunctions()
{
/***************************************/
/*  Add the new Clips features here    */
/*  The function b_ham is called from  */
/*  the command line in CLIPS          */
/***************************************/
extern char *b_ham();
extern char *getgrid();
extern VOID *dummy;
    DefineFunction("b_ham", 's', b_ham, "b_ham");
    DefineFunction("getgrid", 's', getgrid, "getgrid");
}
```

```
/*************************************************************/
/*                        r_plan.c                           */
/*                                                           */
/*  This set of routines provide the route planning mechanism */
/*  to the Automated Scenario Generation System.  These programs */
/*  implement Bresenham's Algorithm to determine grid squares */
/*  along a particular straight line.  As each grid square is */
/*  chosen, its movement factor is examined to see if it is an */
/*  obstacle (movement factor = 6).  If so, then these routines */
/*  first continue to generate grid squares until the far edge of */
/*  the obstacle is found.  Then the routines bisect the line */
/*  through the obstacle and generate the squares on a line which */
/*  passes through the bisection point and is perpendicular to */
/*  the original line of march.  Grid squares are generated until */
/*  the side of the obstacle is found.  The programs then make a */
/*  recursive call to plan the route between the start point and */
/*  the edge of the obstacle and then to plan from the edge of the*/
/*  obstacle to the objective. If another obstacle is encountered,*/
/*  the process is executed again.                           */
/*************************************************************/
/***********************************/
/*          route_planner          */
/*  CLIPS extension version        */
/*                                 */
/*  Date:  12 September 93         */
/*                                 */
/*  Last Change:  12 September 93  */
/*                                 */
/*                                 */
/*  This routine generates terrain */
/*  squares which lie on the line  */
/*  segment which connects the two */
/*  endpoints.  It works for lines */
/*  of any slope in all 4 quadrants.*/
/*                                 */
/*  The program then generates the */
/*  list of terrain squares which  */
/*  are closest to the actual line. */
/*  The routine returns a string of */
/*  route points to CLIPS.         */
/***********************************/
/*****************************************************/
/*  This function uses Bresenham's Algorithm to look  */
/*  at particular grid squares to find obstacles that */
/*  lie on the path from the start to the objective.  */
/*  This function will return true if an obstacle is  */
/*  encountered.                                      */
/*****************************************************/
#include <stdio.h>
#include <math.h>
#include <float.h>
#include <string.h>
#include <stdlib.h>
#include "clips.h"
#include "constant.h"
char *route_planner()
{
FILE *terrain_file_ptr;
```

```
char *returnValue;
char *lexeme1;
char *lexeme2;
char *routelist;
char *temp_route;
char *endptr;
char *avoid_obst();
char grid[9], grid1[9], grid2[9], gridrecord[21];
double  x, x0, x1, y, y0, y1, dy, dx, temp, incr1, incr2, j, k, a, b;
char X0[5], X1[5], Y0[5], Y1[5], Z0[5], M[4];
long offset;
long mobility;
long i;
int secondoctant = 0;

lexeme1 = RtnLexeme(1);
lexeme2 = RtnLexeme(2);
routelist = (char *) malloc(1);
routelist[0] = '\0';
sprintf(grid1, "%-8.8s", lexeme1);
sprintf(grid2, "%-8.8s", lexeme2);

if ((terrain_file_ptr = fopen("terrain.fil", "r")) == NULL) {
 printf("[route_planner] terrain.fil not found\n");
  return("[rotue_planner] Error");
}

  for (i = 0; i < 4; i++) {
X0[i] = grid1[i];
  }
  X0[4] = '\0';

  for (i = 4; i < 8; i++) {
Y0[i - 4] = grid1[i];
  }
  Y0[4] = '\0';

  for (i = 0; i < 4; i++) {
X1[i] = grid2[i];
  }
  X1[4] = '\0';

  for (i = 4; i < 8; i++) {
Y1[i - 4] = grid2[i];
  }
  Y1[4] = '\0';
  x0 = strtod(X0, &endptr);
  y0 = strtod(Y0, &endptr);
  x1 = strtod(X1, &endptr);
  y1 = strtod(Y1, &endptr);

          a = fabs(x1 - x0);
          b = fabs(y1 - y0);
          if (b > a) {
  secondoctant = 1;
  temp = a;
  a = b;
  b = temp;
  }
          temp = 2.0 * b - a;
```

```
  incr1 = 2.0 * (b - a);
  incr2 = 2.0 * b;

/* Slope Conditions */
if ((x0 > x1) && (y0 <= y1))  {
 j = -1.0;
 k = 1.0;
 }
if ((x0 <= x1) && (y0 > y1))  {
j = 1.0;
k = -1.0;
}
if ((x0 <= x1) && (y0 <= y1))  {
j = 1.0;
k = 1.0;
}
if ((x0 > x1) && (y0 > y1))  {
j = -1.0;
k = -1.0;
}
x = x0;
y = y0;
if (secondoctant == 1) {

for (i = 0; i <= a; i++)  {
 sprintf(X0, "%04.0f", x);
 sprintf(Y0, "%04.0f", y);
 sprintf(grid,"%04.0f%s", x,Y0);
 offset = (strtol(grid, &endptr, 10)) * 20;
 fseek(terrain_file_ptr, offset, 0);
 fgets(gridrecord, 21, terrain_file_ptr);
 M[0] = gridrecord[12];
 M[1] = gridrecord[13];
 M[2] = gridrecord[14];
 M[3] = '\0';
 mobility = strtol(M, &endptr, 10);
 if (mobility > 21) {
  routelist[0] ='\0';
  routelist = avoid_obst(grid1, grid2, grid);
  break;
 }
 else {
  Z0[0] = gridrecord[8];
  Z0[1] = gridrecord[9];
  Z0[2] = gridrecord[10];
  Z0[3] = gridrecord[11];
  Z0[4] = '\0';
  routelist = realloc(routelist, strlen(routelist) + strlen(grid)+1);
  strcat(grid, " ");
  strcat(routelist, grid);
  y = y + k;
  if (temp >= 0.0) {
x = x + j;
temp = temp + incr1;
  }
  else
temp = temp + incr2;
  }
  }
}
```

```
  else {

 for (i = 0.0; i <= a; i++) {
 sprintf(X0, "%04.0f", x);
 sprintf(Y0, "%04.0f", y);
 sprintf(grid,"%04.0f%s", x,Y0);
 offset = (strtol(grid, &endptr, 10)) * 20;
 fseek(terrain_file_ptr, offset, 0);
 fgets(gridrecord, 21, terrain_file_ptr);
 M[0] = gridrecord[12];
 M[1] = gridrecord[13];
 M[2] = gridrecord[14];
 M[3] = '\0';
 mobility = strtol(M, &endptr, 10);
 if (mobility > 21) {
  routelist[0] ='\0';

  routelist = avoid_obst(grid1, grid2, grid);
  break;
 }
 else {
  Z0[0] = gridrecord[8];
  Z0[1] = gridrecord[9];
  Z0[2] = gridrecord[10];
  Z0[3] = gridrecord[11];
  Z0[4] = '\0';
  routelist = realloc(routelist, strlen(routelist) + strlen(grid)+1);
  strcat(grid, " ");
  strcat(routelist, grid);
  x = x + j;
  if (temp >= 0.0) {
y = y + k;
temp = temp + incr1;
  }
  else
temp = temp + incr2;
}
  }
  }

fclose(terrain_file_ptr);
returnValue = AddSymbol(routelist);
return(returnValue);
}
/************************************/
/*        route_planner1.c          */
/*        CLIPS extension           */
/*       recursive version          */
/*                                  */
/*  Date:  14 July 93               */
/*                                  */
/*  Last Change:  14 July 93        */
/*                                  */
/*                                  */
/*  This routine generates terrain  */
/*  squares which lie on the line   */
/*  segment which connects the two  */
/*  endpoints.  It works for lines  */
/*  of any slope in all 4 quadrants.*/
```

```
/*                                  */
/*  The program then generates the  */
/*  list of terrain squares which   */
/*  are closest to the actual line. */
/*  The routine returns a string of */
/*  route points to CLIPS.          */
/************************************/
/*******************************************************/
/*  This function uses Bresenham's Algorithm to look  */
/*  at particular grid squares to find obstacles that */
/*  lie on the path from the start to the objective.  */
/*  This function will return true if an obstacle is  */
/*  encountered.                                      */
/*******************************************************/
#include <stdio.h>
#include <math.h>
#include <float.h>
#include <string.h>
#include <stdlib.h>
#include "clips.h"
#include "constant.h"
char *route_planner1(char *lexeme1, char *lexeme2)
{
FILE *terrain_file_ptr;
char *returnValue;
char *routelist;
char *temp_route;
char *endptr;
char *avoid_obst();
char grid[9], grid1[9], grid2[9], gridrecord[21];
double  x, x0, x1, y, y0, y1, dy, dx, temp, incr1, incr2, j, k, a, b;
char X0[5], X1[5], Y0[5], Y1[5], Z0[5], M[4];
long offset;
long mobility;
long i;
int secondoctant = 0;
routelist = (char *) malloc(1);
routelist[0] = '\0';
sprintf(grid1, "%-8.8s", lexeme1);
sprintf(grid2, "%-8.8s", lexeme2);
if ((terrain_file_ptr = fopen("terrain.fil", "r")) == NULL) {
 printf("[route_planner1] terrain.fil not found\n");
  return("[route_planner1] Error");
}

X0[i] = grid1[i];
  for (i = 0; i < 4; i++) {
  }
  X0[4] = '\0';

  for (i = 4; i < 8; i++) {
Y0[i - 4] = grid1[i];
  }
  Y0[4] = '\0';

  for (i = 0; i < 4; i++) {
X1[i] = grid2[i];
  }
  X1[4] = '\0';
```

```
  for (i = 4; i < 8; i++) {
Y1[i - 4] = grid2[i];
  }
  Y1[4] = '\0';


  x0 = strtod(X0, &endptr);

  y0 = strtod(Y0, &endptr);

  x1 = strtod(X1, &endptr);

  y1 = strtod(Y1, &endptr);

  a = fabs(x1 - x0);
  b = fabs(y1 - y0);
  if (b > a) {
  secondoctant = 1;
  temp = a;
  a = b;
  b = temp;
  }
  temp = 2.0 * b - a;
  incr1 = 2.0 * (b - a);
  incr2 = 2.0 * b;

/* Slope Conditions */
if ((x0 > x1) && (y0 <= y1))  {
 j = -1.0;
 k = 1.0;
 }
if ((x0 <= x1) && (y0 > y1))  {
j = 1.0;
k = -1.0;
}
if ((x0 <= x1) && (y0 <= y1))  {
j = 1.0;
k = 1.0;
}
if ((x0 > x1) && (y0 > y1))  {
j = -1.0;
k = -1.0;
}
x = x0;
y = y0;
if (secondoctant == 1) {

for (i = 0; i <= a; i++)  {
 sprintf(X0, "%04.0f", x);
 sprintf(Y0, "%04.0f", y);
 sprintf(grid,"%04.0f%s", x,Y0);
 offset = (strtol(grid, &endptr, 10)) * 20;
 fseek(terrain_file_ptr, offset, 0);
 fgets(gridrecord, 21, terrain_file_ptr);
 M[0] = gridrecord[12];
 M[1] = gridrecord[13];
 M[2] = gridrecord[14];
 M[3] = '\0';
 mobility = strtol(M, &endptr, 10);
 if (mobility > 21) {
```

```
  routelist[0] ='\0';
  routelist = avoid_obst(grid1, grid2, grid);
  break;
 }
 else {
  Z0[0] = gridrecord[8];
  Z0[1] = gridrecord[9];
  Z0[2] = gridrecord[10];
  Z0[3] = gridrecord[11];
  Z0[4] = '\0';
  routelist = realloc(routelist, strlen(routelist) + strlen(grid)+1);
  strcat(grid, " ");
  strcat(routelist, grid);
  y = y + k;
  if (temp >= 0.0) {
x = x + j;
temp = temp + incr1;
  }
  else
temp = temp + incr2;
  }
  }
}

  else {

 for (i = 0.0; i <= a; i++) {
 sprintf(X0, "%04.0f", x);
 sprintf(Y0, "%04.0f", y);
 sprintf(grid,"%04.0f%s", x,Y0);
 offset = (strtol(grid, &endptr, 10)) * 20;
 fseek(terrain_file_ptr, offset, 0);
 fgets(gridrecord, 21, terrain_file_ptr);
 M[0] = gridrecord[12];
 M[1] = gridrecord[13];
 M[2] = gridrecord[14];
 M[3] = '\0';
 mobility = strtol(M, &endptr, 10);
 if (mobility > 21) {
  routelist[0] ='\0';

  routelist = avoid_obst(grid1, grid2, grid);
  break;
 }
 else {
  Z0[0] = gridrecord[8];
  Z0[1] = gridrecord[9];
  Z0[2] = gridrecord[10];
  Z0[3] = gridrecord[11];
  Z0[4] = '\0';
  routelist = realloc(routelist, strlen(routelist) + strlen(grid)+1);
  strcat(grid, " ");
  strcat(routelist, grid);
  x = x + j;
  if (temp >= 0.0) {
y = y + k;
temp = temp + incr1;
  }
  else
temp = temp + incr2;
```

```
}
  }
  }

fclose(terrain_file_ptr);
return (routelist);
}
/************************************/
/*          avoid.c                 */
/*  CLIPS extension version         */
/*                                  */
/*  Date:  14 July 93               */
/*                                  */
/*  Last Change:  14 July 93        */
/*                                  */
/*                                  */
/*  This routine generates terrain  */
/*  squares which avoid obstacles.  */
/*                                  */
/*  The program then generates the  */
/*  list of terrain squares which   */
/*  are closest to the actual line. */
/*  The routine returns a string of */
/*  route points to CLIPS.          */
/************************************/
/*********************************************************/
/*  This function uses Bresenham's Algorithm to look  */
/*  at particular grid squares to find obstacles that */
/*  lie on the path from the start to the objective.  */
/*  This function will return true if an obstacle is  */
/*  encountered.                                      */
/*********************************************************/
#include <stdio.h>
#include <math.h>
#include <float.h>
#include <string.h>
#include <stdlib.h>
#include "clips.h"
#include "constant.h"

char *avoid_obst(char *point1, char *point2, char *point3)
{
FILE *terrain_file_ptr;
        char *far_edge();
char *route_planner1();
char *returnValue;
char *routelist, *backedge, *first_leg, *second_leg;
char temp_point[9];
char *endptr, *endptr1;
char middle[9], X0[5], Y0[5], X1[5], Y1[5], X2[5], Y2[5], X3[5], Y3[5];
char M[5], gridrecord[21], Z0[5], grid[5], backedge1[9];
double slope, x,x0,x1,y,y0,y1,x2, y2, x3, y3;
double dy,dx,temp,incr1,incr2,k,a,b;
double xtemp, ytemp, lcount, rcount;
long offset;
long rmobility, lmobility;
long mobility = 61;
long i;
int j;
int rflag, lflag;
```

```
int secondoctant = 0;
int islope;

routelist = malloc(1);
routelist[0] ='\0';
        backedge = far_edge(point3, point2);
strcpy(backedge1, backedge);
backedge1[8] = '\0';

if ((terrain_file_ptr = fopen("terrain.fil", "r")) == NULL) {
 printf("[avoid_obst] terrain.fil not found\n");
  return("[avoid_obst] Error");
}

for (i = 0; i < 4; i++) {
X0[i] = point3[i];
}
X0[4] = '\0';

for (i = 4; i < 8; i++) {
Y0[i - 4] = point3[i];
}
Y0[4] = '\0';

for (i = 0; i < 4; i++) {
X1[i] = backedge1[i];
}
X1[4] = '\0';

for (i = 4; i < 8; i++) {
Y1[i - 4] = backedge1[i];
}
Y1[4] = '\0';

for (i = 0; i < 4; i++) {
X2[i] = point1[i];
}
X2[4] = '\0';

for (i = 4; i < 8; i++) {
Y2[i - 4] = point1[i];
}
Y2[4] = '\0';

for (i = 0; i < 4; i++) {
X3[i] = point2[i];
}
X3[4] = '\0';

for (i = 4; i < 8; i++) {
Y3[i - 4] = point2[i];
}
Y3[4] = '\0';

x0 = strtod(X0, &endptr);
y0 = strtod(Y0, &endptr);
x1 = strtod(X1, &endptr);
y1 = strtod(Y1, &endptr);

/*  Save original points for later calculations  */
```

```
x2 = strtod(X2, &endptr);
y2 = strtod(Y2, &endptr);
x3 = strtod(X3, &endptr);
y3 = strtod(Y3, &endptr);

if ((((x3 - x2) < 0.0) && ((y3 - y2) > 0.0)) || (((x3 - x2) > 0.0) && ((y3 - y2) < 0.0)
islope = 1;
 }
 else islope = -1;


dx = x1 - x0;
dy = y1 - y0;

/*    Find the Middle Point of the Line segment crossing the obstacle  */
if ((x1 >= x0) && (y1 >= y0)) {
xtemp = floor((x1 - x0) / 21) + x0;
ytemp = floor((y1 - y0) / 21) + y0;
}
 else {
xtemp = ceil((x1 - x0) / 21) + x0;
ytemp = ceil((y1 - y0) / 21) + y0;
}

sprintf(middle, "%04.0f%04.0f\0", xtemp, ytemp);

x = xtemp;
y = ytemp;
lcount = 0;
rcount = 0;
j = 0;


/* generate grid squares along the perpendicular line */

 while (mobility > 21) {
 j++;

 /* right square */
 x = xtemp;
 y = ytemp;

 if ((dy == 0.0) && (x0 != x1))  {
  rcount++;
 x = xtemp;
  y = ytemp - rcount;
  }
 else {
  if ((dx == 0.0) && (y0 != y1)){
 rcount++;
 x = xtemp + rcount;
 y = ytemp;
 }
 else {
  if ((fabs (dx) <= fabs(dy)) && (x1 != x0) && (y1 != y0)) {
 rcount++;
 x = xtemp + rcount;
y = ytemp + (-dx /dy) * (x - xtemp);
  }
  else {
```

```
    if ((x1 != x0) && (y1 != y0))  {
 rcount++;
 y = ytemp - rcount;
 x = (-dy / dx) * (y - ytemp) + xtemp;
}
else {
 if (islope == 1) {
rcount ++;
x = xtemp + rcount;
y = ytemp + rcount;
 }
 else {
  rcount ++;
  x = xtemp + rcount;
  y = ytemp - rcount;
 }
}

 }
}
 }


 if (x < 0.0) x = 0.0;
        if (y < 0.0) y = 0.0;
    if (fabs(dx) <= fabs (dy)) {
x = floor(x + 0.5);
y = floor(y);
  }
  else {
    x = floor(x + 0.5);
    y = floor(y + 0.5);
  }

 sprintf(X0, "%04.0f", x);
 sprintf(Y0, "%04.0f", y);
 sprintf(grid,"%04.0f%s", x,Y0);
 offset = (strtol(grid, &endptr, 10)) * 20;
 fseek(terrain_file_ptr, offset, 0);
 fgets(gridrecord, 21, terrain_file_ptr);
 M[0] = gridrecord[12];
 M[1] = gridrecord[13];
 M[2] = gridrecord[14];
 M[3] = '\0';
 rmobility = strtol(M, &endptr, 10);
  if (rmobility < 61) break;


 /* left square  */
if ((dy == 0.0) && (x0 != x1)) {
 x = xtemp;
 lcount++;
 y = ytemp + lcount;
 }
  else {
 if ((dx == 0.0) && (y0 != y1)){
 lcount++;
 x = xtemp - lcount;
 y = ytemp;
  }
```

```
 else {
  if (fabs(dx) <= fabs (dy) && (x0 != x1) && (y0 != y1)) {
 lcount++;
 x = xtemp - lcount;
 y = (- dx / dy) * (x - xtemp) + ytemp;
}
else {
 if ((x1 != x0) && (y0 != y1)) {
  lcount++;
  y = ytemp + lcount;
  x = (- dy / dx) * (y - ytemp) + xtemp;
 }
 else {
   if (islope == 1)  {
        lcount++;
x = xtemp - lcount;
y = ytemp - lcount;
  }
    else {
     lcount++;
     x = xtemp - lcount;
     y = ytemp + lcount;
     }
}
 }
}
  }
  if (x < 0.0) x = 0.0;
  if (y < 0.0) y = 0.0;
  if (fabs(dx) <= fabs(dy)) {
  x = floor(x + 0.5);
  y = floor(y);
  }
  else {
      x = floor(x + 0.5);
    y = floor(y + 0.5);
  }
 sprintf(X0, "%04.0f", x);
 sprintf(Y0, "%04.0f", y);
 sprintf(grid,"%04.0f%s", x,Y0);
 offset = (strtol(grid, &endptr, 10)) * 20;
 fseek(terrain_file_ptr, offset, 0);
 fgets(gridrecord, 21, terrain_file_ptr);
 M[0] = gridrecord[12];
 M[1] = gridrecord[13];
 M[2] = gridrecord[14];
 M[3] = '\0';
 lmobility = strtol(M, &endptr, 10);
 if (lmobility < 61) break;
}
routelist = realloc(routelist, 1);
routelist[0] = '\0';
first_leg = route_planner1(point1, grid);
second_leg = route_planner1(grid, point2);
routelist = realloc(routelist, strlen(routelist) + strlen(first_leg) + 1);
strncat(routelist, first_leg, strlen(first_leg) - 9);
routelist = realloc(routelist, strlen(routelist) + strlen(second_leg) + 1);
strcat(routelist, second_leg);
fclose(terrain_file_ptr);
```

```
return (routelist);
}
/*************************************/
/*          far_edge.c               */
/*  CLIPS extension version          */
/*                                   */
/*  Date:  14 July 93                */
/*                                   */
/*  Last Change:  14 July 93         */
/*                                   */
/*                                   */
/*  This routine generates terrain   */
/*  squares which lie on the line    */
/*  segment which connects the two   */
/*  endpoints.  It works for lines   */
/*  of any slope in all 4 quadrants. */
/*                                   */
/*  The program returns grid point   */
/* which lies on the back edge of    */
/* a given obstacle.                 */
/*************************************/
#include <stdio.h>
#include <math.h>
#include <float.h>
#include <string.h>
#include <stdlib.h>
#include "clips.h"
#include "constant.h"
char *far_edge(char *point1, char *point2)
{
FILE *terrain_file_ptr;
char *returnValue;
char *lexeme1;
char *lexeme2;
char *routelist;
char *temp_route;
char *endptr;
char grid[9], grid1[9], grid2[9], gridrecord[21], previous[9];
double  x, x0, x1, y, y0, y1, dy, dx, temp, incr1, incr2, j, k, a, b;
char X0[5], X1[5], Y0[5], Y1[5], Z0[5], M[4];
long offset;
long mobility;
long i;
int secondoctant = 0;
lexeme1 = malloc(strlen(point1));
        lexeme2 = malloc(strlen(point2));
sprintf(lexeme1,"%-8.8s",point1);
sprintf(lexeme2,"%-8.8s",point2);
routelist = (char *) malloc(1);
routelist[0] = '\0';
sprintf(grid1, "%-8.8s", lexeme1);
sprintf(grid2, "%-8.8s", lexeme2);

if ((terrain_file_ptr = fopen("terrain.fil", "r")) == NULL) {
 printf("[far_edge] terrain.fil not found\n");
  return("[far_edge] Error");
 }

for (i = 0; i < 4; i++) {
X0[i] = grid1[i];
```

```
}
X0[4] = '\0';

for (i = 4; i < 8; i++) {
Y0[i - 4] = grid1[i];
}
Y0[4] = '\0';

for (i = 0; i < 4; i++) {
X1[i] = grid2[i];
}
X1[4] = '\0';

for (i = 4; i < 8; i++) {
Y1[i - 4] = grid2[i];
}
Y1[4] = '\0';

x0 = strtod(X0, &endptr);
y0 = strtod(Y0, &endptr);
x1 = strtod(X1, &endptr);
y1 = strtod(Y1, &endptr);
        a = fabs(x1 - x0);
        b = fabs(y1 - y0);
        if (b > a) {
  secondoctant = 1;
  temp = a;
  a = b;
  b = temp;
  }
        temp = 2.0 * b - a;
        incr1 = 2.0 * (b - a);
        incr2 = 2.0 * b;

/* Slope Conditions */
if ((x0 > x1) && (y0 <= y1))  {
 j = -1.0;
 k = 1.0;
 }
if ((x0 <= x1) && (y0 > y1))  {
j = 1.0;
k = -1.0;
}
if ((x0 <= x1) && (y0 <= y1))  {
j = 1.0;
k = 1.0;
}
if ((x0 > x1) && (y0 > y1))  {
j = -1.0;
k = -1.0;
}
x = x0;
y = y0;
if (secondoctant == 1) {
for (i = 0; i <= a; i++)  {
sprintf(X0, "%04.0f", x);
sprintf(Y0, "%04.0f", y);
sprintf(grid,"%04.0f%s", x,Y0);
offset = (strtol(grid, &endptr, 10)) * 20;
fseek(terrain_file_ptr, offset, 0);
```

```
fgets(gridrecord, 21, terrain_file_ptr);
M[0] = gridrecord[12];
M[1] = gridrecord[13];
M[2] = gridrecord[14];
M[3] = '\0';
mobility = strtol(M, &endptr, 10);
if (mobility < 61) break;
y = y + k;
if (temp >= 0.0) {
  x = x + j;
  temp = temp + incr1;
}
else
temp = temp + incr2;
sprintf(previous, "%-8.8s", grid);
}
}
else  {
 for (i = 0.0; i <= a; i++) {
 sprintf(X0, "%04.0f", x);
 sprintf(Y0, "%04.0f", y);
 sprintf(grid,"%04.0f%s", x,Y0);
 offset = (strtol(grid, &endptr, 10)) * 20;
 fseek(terrain_file_ptr, offset, 0);
 fgets(gridrecord, 21, terrain_file_ptr);
 M[0] = gridrecord[12];
 M[1] = gridrecord[13];
 M[2] = gridrecord[14];
 M[3] = '\0';
 mobility = strtol(M, &endptr, 10);
 if (mobility < 61) break;
 x = x + j;
 if (temp >= 0.0) {
 y = y + k;
 temp = temp + incr1;
}
 else
 temp = temp + incr2;
  sprintf(previous, "%-8.8s", grid);

  }
}


fclose(terrain_file_ptr);
return (previous);
}
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;    Automated Scenario Generation System (ASGS)   ;;
;;            Interface Program                     ;;
;;                                                  ;;
;;   Author:  CPT Mark W. Pfefferman                ;;
;;     Date:  28 August 93                          ;;
;;  File name: asgs.clp                             ;;
;;  Purpose:  This CLIPS Program is the interface   ;;
;;  for adding new missions and entities to the     ;;
;;  system.  To add an entity, the user edits the   ;;
;;  the entity.clp program.  The structure of the   ;;
;;  entity attributes is the BATTLESIM format as    ;;
;;  described by Bergman's thesis.  To add a new    ;;
;;  mission, the user edits the mission.clp.        ;;
;;  EXECUTION:  To run the Automated Scenario       ;;
;;  Generation System (ASGS) follow these steps:    ;;
;;   1. clips                                       ;;
;;   2. (load asgs.clp)                             ;;
;;   3. (asgs)                                      ;;
;;   4. (execute-mission)                           ;;
;;  Type these commands exactly as shown, and the   ;;
;;  system will generate a sceanrio file based on   ;;
;;  the contents of the mission.fil file            ;;
;;  ASSUMPTIONS:                                    ;;
;;  1.  Only 5 icon definitions have been defined   ;;
;;      for BATTLESIM so far.  The assumption is    ;;
;;      that the numbering of the icon definitions  ;;
;;      will remain as currently used by BATTLESIM  ;;
;;      and presented in Bergman's Thesis:          ;;
;;     1 - f18                                      ;;
;;     2 - mig1                                     ;;
;;     3 - missile                                  ;;
;;     4 - tank                                     ;;
;;     5 - truck                                    ;;
;;  2.  Changes to this numbering scheme will cause ;;
;;      adjustments to be made in the slot ''id''   ;;
;;      default values in the objects.              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deffunction asgs ()
  (load entity.clp)
  (load driver.clp)
  (load mission_definitions.clp)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   Automated Scenario Generation System (ASGS)    ;;
;;                Driver Program                    ;;
;;                                                  ;;
;;   Author:  CPT Mark W. Pfefferman                ;;
;;     Date:  14 November 93                        ;;
;;  File name: driver.clp                           ;;
;;                                                  ;;
;;  Purpose:  This CLIPS Program  is the driver for ;;
;;  the ASGS.  After loading and running asgs.clp,  ;;
;;  the user types (execute-mission) which starts   ;;
;;  the scenario generation process.  The mission   ;;
;;  file is read and parsed.  Entities are created  ;;
;;  based on the mission file and are assigned a    ;;
;;  mission to plan.  The results of this planning  ;;
;;  are written to a scenario file in the format    ;;
;;  required by BATTLESIM.                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Define some global variables to hold information    ;;
;;  needed by other functions throughout the reasoning  ;;
;;  process.                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defglobal ?*route_list* = ""
           ?*temp_route* = ""
           ?*who* = ""
           ?*what* =""
           ?*when* =""
           ?*class* =""
           ?*nomenclature* =""
           ?*mission* = ""
           ?*start* = ""
           ?*objective* = ""
           ?*cp1* = ""
           ?*cp2* = ""
           ?*cp3* = "")

(defglobal ?*grid* = ""
           ?*grid_record* = ""
           ?*cp_count* = 0
           ?*distance* = 0
           ?*count* = 0
           ?*object_list* =""
           ?*id* = ""
           ?*temp* = 0)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Function to instantiate an entity  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deffunction create-entity (?type
                            ?nomenclature
                            ?bumper_number
                           )
   (make-instance ?bumper_number of ?type
      (nomenclature        ?nomenclature)
      (bumper_number       ?bumper_number)
   )
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;  Print Scenario File Battlefield Information  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deffunction print-battlefield_info()
   (open "scenario.fil" outfile "a")
   (open "scenhead.fil" infile2 "r")
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Read in version number from BATTLESIM file  ;;
;; and write it to the scenario file.          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   (printout outfile "* version number" crlf)
   (printout outfile (readline infile2) crlf)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Read in terrain file name and write it  ;;
;;  to the scenario file.                   ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   (printout outfile "* terrain data filename" crlf)
   (printout outfile (readline infile2) crlf)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Read in battlefield sector information  ;;
;;  write it to the scenario file.          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   (printout outfile "* terrain min coordinates (x, y, z)" crlf)
   (printout outfile (readline infile2) crlf)
   (printout outfile "* terrain max coordinates (x, y, z)" crlf)
   (printout outfile (readline infile2) crlf)
   (printout outfile "* number of sectors (must be < 64)" crlf)
   (printout outfile (readline infile2) crlf)
   (printout outfile
   "* sector min/max boundaries (x, y, z values in order from 1st to last sectors)" crl
   (printout outfile (readline infile2) crlf)
   (printout outfile (readline infile2) crlf)
   (printout outfile (readline infile2) crlf)
   (printout outfile (readline infile2) crlf)
   (printout outfile (readline infile2) crlf)
   (printout outfile (readline infile2) crlf)
   (printout outfile (readline infile2) crlf)
   (printout outfile (readline infile2) crlf)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Currently five icons are supported by BATTLESIM  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   (printout outfile "* number of icon records" crlf)
   (printout outfile "5" crlf)
   (printout outfile "1 f18" crlf)
   (printout outfile "2 mig1" crlf)
   (printout outfile "3 missile" crlf)
   (printout outfile "4 tank" crlf)
   (printout outfile "5 truck" crlf)
;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Close all the files  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;
   (close outfile)
   (close infile2)
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Put return string of route points in proper format   ;;
;;  and write it to a plot file and to the scenario file ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deffunction write_route ()
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;  Count Number of route points ;;
;;  in route.                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   (bind ?count 1)
   (bind ?*temp* 0)
   (while (< ?count (str-length ?*route_list*))
       (bind ?*temp* (+ ?*temp* 1))
       (bind ?count (+ ?count 9))
   )
   (printout outfile "* number of route points " crlf)
   (printout outfile ?*temp* crlf)
   (printout outfile "* route coordinates x, y, z (start to finish order)" crlf)
   (bind ?count 1)
   (while (< ?count (str-length ?*route_list*))
       (printout plotfile (float (eval
                          (sub-string ?count (+ ?count 3)
       ?*route_list*))) "  "
                          (float (eval
                           (sub-string (+ ?count 4) (+ ?count 7)
       ?*route_list*)))  crlf)

       (printout outfile (float (eval
                          (sub-string ?count (+ ?count 3) ?*route_list*))) ", "
                         (float (eval
                          (sub-string (+ ?count 4) (+ ?count 7) ?*route_list*)))
                          ;;; Z coordinate hard coded at 1000 ;;;;;;;
                          ", 1000.0" crlf)
       (bind ?count (+ ?count 9))
   )
   (printout outfile crlf)
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Read Mission List, parse out the key fields and execute  ;;
;;  the mission.                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deffunction execute-mission()
   (print-battlefield_info)
   (open "mission.fil" infile)
   (bind ?mission_string (readline infile))
   (while (!= (str-compare ?mission_string EOF) 0)
      (bind $?mission_strings (str-explode ?mission_string))
      (printout t "Mission_strings =" $?mission_strings crlf)
      (bind ?*id* (nth 1 $?mission_strings))
      (bind ?*who* (symbol-to-instance-name (nth 2 $?mission_strings)))
      (bind ?*class* (eval (nth 3 $?mission_strings)))
      (bind ?*nomenclature* (eval (nth 4 $?mission_strings)))
      (bind ?*what* (eval (nth 5  $?mission_strings)))
      (bind ?*when* (eval (nth 6 $?mission_strings)))
      (bind ?*start* (sub-string 1 8
                     (getgrid (nth 7 $?mission_strings))))
      (bind ?*objective* (sub-string 1 8
                         (getgrid (nth 8 $?mission_strings))))
      (if (!= (str-compare (str-cat (nth 9 $?mission_strings)) nil) 0)
          then
            (bind ?*cp1* (sub-string 1 8
                         (getgrid (nth 9 $?mission_strings))))
            (bind ?*cp_count* 1)
            (assert (mode checkpoint))
          else
            (assert (mode automatic))
```

```
      )
      (if (!= (str-compare (str-cat (nth 10 $?mission_strings)) nil) 0)
          then
            (bind ?*cp2* (sub-string 1 8
                           (getgrid (nth 10 $?mission_strings))))
            (bind ?*cp_count* 2)
      )
      (if (!= (str-compare (str-cat (nth 11 $?mission_strings)) nil) 0)
          then
            (bind ?*cp3* (sub-string 1 8
                           (getgrid (nth 11 $?mission_strings))))
            (bind ?*cp_count* 3)
      )

      (bind ?file (str-cat ?*who* ".plt"))
      (open ?file plotfile "w")
      (open "scenario.fil" outfile "a")
      (assert (phase notover ?*who*))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Instantiate an object of a particular class  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      (create-entity ?*class* ?*nomenclature* ?*who*)

      (printout outfile "* object type thru max climb" crlf)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Assign the line nubmer of the object in the mission file    ;;
;;  to the object as its id (unique ids required by BATTLESIM)  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      (send ?*who* put-id ?*id*)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Now build the object type thru max climb string using the  ;;
;;  the actual slot values of the instantiated object.          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      (printout outfile (send ?*who* get-type) " ")
      (printout outfile (send ?*who* get-id) " ")
      (printout outfile "0 ")
      (printout outfile (float(eval(sub-string 1 4 ?*start*))) " ")
      (printout outfile (float(eval(sub-string 5 8 ?*start*))) " ")
      (printout outfile "1000.0 ")
      (printout outfile (send ?*who* get-x_velocity) " ")
      (printout outfile (send ?*who* get-y_velocity) " ")
      (printout outfile (send ?*who* get-z_velocity) " ")
      (printout outfile (send ?*who* get-yaw_rate) " ")
      (printout outfile (send ?*who* get-pitch_rate) " ")
      (printout outfile (send ?*who* get-roll_rate) " ")
      (printout outfile (send ?*who* get-size) " ")
      (printout outfile (send ?*who* get-mass) " ")
      (printout outfile crlf)
      (printout outfile (send ?*who* get-loyalty) " ")
      (printout outfile (send ?*who* get-fuel_status) " ")
      (printout outfile (send ?*who* get-condition) " ")
      (printout outfile (send ?*who* get-vulnerability) " ")
      (printout outfile (send ?*who* get-experience) " ")
      (printout outfile (send ?*who* get-threat_knowledge) " ")
      (printout outfile (send ?*who* get-min_turn_rad) " ")
      (printout outfile (send ?*who* get-speed) " ")
      (printout outfile (send ?*who* get-avg_fuel_consump) " ")
      (printout outfile (send ?*who* get-max_climb) " " crlf)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;  Tell the object what its mission is and its key terrain points   ;;
;;  are.  The object will then perform the actions determined by the ;;
;;  message-handler and the rules in mission_definitions.clp         ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      (send ?*who* ?*what* ?*start* ?*objective*)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  After the object has completed planning its mission, the  ;;
;;  progrma will return to this point and write the rest of   ;;
;;  the required object data to the scenario file.  This data ;;
;;  is gotten from the actual slot values of the instantiated ;;
;;  object.                                                   ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      (printout outfile "* number of sensors" crlf)
      (printout outfile (send ?*who* get-sensors) crlf)
      (printout outfile (send ?*who* get-sensor_type) " ")
      (printout outfile (send ?*who* get-sensor_range) " ")
      (printout outfile (send ?*who* get-sensor_resolution) crlf)
      (printout outfile "* number of armaments" crlf)
      (printout outfile (send ?*who* get-armaments) crlf)
      (printout outfile "* armaments description (if above > 0)" crlf)
      (printout outfile (send ?*who* get-armament_description) crlf)
      (printout outfile "* number of targets" crlf)
      (printout outfile (send ?*who* get-targets) crlf)
      (printout outfile "* target descriptions (if above > 0)" crlf)
      (printout outfile (send ?*who* get-target_list) crlf)
      (printout outfile "* number of defensive systems" crlf)
      (printout outfile (send ?*who* get-defensive_systems) crlf)
      (printout outfile "* defensive systems descriptions (if above > 0)"  crlf)
      (printout outfile (send ?*who* get-defensive_sys_descr) crlf)
      (printout outfile "* END OF OBJECT" crlf)
      (close outfile)
      (close plotfile)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Reset buffer values for the next mission ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      (bind ?*route_list* "")
      (bind ?*distance* 0)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Read the next mission from the mission file  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      (bind ?mission_string (readline infile))
)
  (close infile)
  (halt)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;    Automated Scenario Generation System (ASGS)                          ;;
;;              Mission Rules                                              ;;
;;                                                                         ;;
;;   Author:  CPT Mark W. Pfefferman                                       ;;
;;     Date:   18 October 1993                                             ;;
;;  File name: mission_definitions.clp                                     ;;
;;                                                                         ;;
;;  Purpose:  This CLIPS Program provides the rules                        ;;
;;  for missions which the entities will perform in                        ;;
;;  generating a scenario.  The user can add more                          ;;
;;  missions to this rule base by defining a message                       ;;
;;  handler and then by including the rules required                       ;;
;;  to plan or conduct that mission.  Two missions                         ;;
;;  included here:  route-recon and bomb-target.                           ;;
;;  Both of tnese missions call the route planning                         ;;
;;  function to determine routes avoiding obstacles.                       ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Method that the object uses to recon a route                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defmessage-handler base-entity route-recon (?start ?objective)
(printout t "Reconing Route " ?start crlf)
(bind ?*start* ?start)
(bind ?*grid* ?start)
(bind ?*objective* ?objective)
(assert (phase notover ?*who*))
(assert (travel one-way))
(run)
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Method that the object uses to bomb a target                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defmessage-handler plane bomb-target (?start ?objective)
(printout t "Bombing Target " ?start crlf)
(bind ?*start* ?start)
(bind ?*grid* ?start)
(bind ?*objective* ?objective)
(assert (phase notover ?*who*))
(assert (travel two-way))
(run)
  )

;*********************************************
;*****     Rules to plan out a route     *****
;*********************************************
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  No checkpoints included                 ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defrule add-points
?f1 <- (phase notover ?entity)
?f2 <- (mode automatic)
=>
(bind ?*route_list* (str-cat ?*route_list*
    (route_planner ?*start* ?*objective*)))
(retract ?f1)
```

```
(assert (phase over ?entity))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Mandatory Checkpoints have been included  ;;
;;  Used in one way type missions like route- ;;
;;  recon.                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defrule route-with-cp1
  ?f1 <- (phase notover ?entity)
  ?f2 <- (mode checkpoint)
  ?f3 <- (travel one-way)
  =>
  (printout t "Made it to route-with-cp1" crlf)
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;  If only one check point  ;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  (if (= ?*cp_count* 1)
 then
 (bind ?*route_list* (route_planner ?*start* ?*cp1*))
 (bind ?*temp_route* (route_planner ?*cp1* ?*objective*))
 (bind ?*route_list* (str-cat ?*route_list*
(sub-string 10 (str-length ?*temp_route*)
        ?*temp_route*)))
  )
  ;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;  If two checkpoints  ;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;
  (if (= ?*cp_count* 2)
then
(bind ?*route_list* (route_planner ?*start* ?*cp1*))
(bind ?*temp_route* (route_planner ?*cp1* ?*cp2*))
(bind ?*route_list* (str-cat ?*route_list*
    (sub-string 10 (str-length ?*temp_route*)
       ?*temp_route*)))
(bind ?*temp_route* (route_planner ?*cp2* ?*objective*))
(bind ?*route_list* (str-cat ?*route_list*
    (sub-string 10 (str-length ?*temp_route*)
       ?*temp_route*)))
  )
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;  If three checkpoints  ;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;
  (if (= ?*cp_count* 3)
then
(bind ?*route_list* (route_planner ?*start* ?*cp1*))
(bind ?*temp_route* (route_planner ?*cp1* ?*cp2*))
(bind ?*route_list* (str-cat ?*route_list*
    (sub-string 10 (str-length ?*temp_route*)
       ?*temp_route*)))
(bind ?*temp_route* (route_planner ?*cp2* ?*cp3*))
(bind ?*route_list* (str-cat ?*route_list*
    (sub-string 10 (str-length ?*temp_route*)
       ?*temp_route*)))
(bind ?*temp_route* (route_planner ?*cp3* ?*objective*))
(bind ?*route_list* (str-cat ?*route_list*
    (sub-string 10 (str-length ?*temp_route*)
       ?*temp_route*)))
 )
 (write_route)
```

```
 (retract ?f1 ?f2 ?f3)
 (assert (phase over ?entity))
 (assert (travel one-way))
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Mandatory Checkpoints have been included  ;;
;;  Used in two-way missions like bomb-target ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defrule route-with-cp2
  ?f1 <- (phase notover ?entity)
  ?f2 <- (mode checkpoint)
  ?f3 <- (travel two-way)
  =>
  (printout t "Made it to route-with-cp2" crlf)
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;  Plan Route to objective  ;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  (bind ?*route_list* (route_planner ?*start* ?*cp1*))
  (bind ?*temp_route* (route_planner ?*cp1* ?*objective*))
  (bind ?*route_list* (str-cat ?*route_list*
(sub-string 10 (str-length ?*temp_route*)
?*temp_route*)))
  (bind ?*temp_route* (route_planner ?*objective* ?*cp2*))
  (bind ?*route_list* (str-cat ?*route_list*
      (sub-string 10 (str-length ?*temp_route*)
 ?*temp_route*)))
(bind ?*temp_route* (route_planner ?*cp2* ?*start*))
(bind ?*route_list* (str-cat ?*route_list*
    (sub-string 10 (str-length ?*temp_route*)
       ?*temp_route*)))
  (retract ?f1 ?f3)
  (assert (phase over ?entity))
  (assert (travel one-way))
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Stop when you complete the mission ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defrule mission-complete2
?f1 <- (phase over ?entity)
?f2 <- (travel two-way)
=>
(printout t ?*who* " has reached the objective " ?*what* " mission." crlf)
(bind ?*grid* ?*objective*)
(bind ?*objective* ?*start*)
(bind ?*start* ?*grid*)
(retract ?f1 ?f2)
(assert (phase notover ?entity))
(assert (travel one-way))
(bind ?*temp_route* ?*route_list*)
(bind ?*route_list* (sub-string 1 (- (str-length ?*temp_route*) 9)
?*temp_route*))
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Stop when you complete the mission ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defrule mission-complete1
(phase over ?entity)
(travel one-way)
=>
(printout t ?*who* " has completed its " ?*what* " mission." crlf)
```

```
(write_route)
(retract *)
  )
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;               Entity Descriptions               ;;
;;                                                 ;;
;;   Author:  CPT Mark W. Pfefferman               ;;
;;    Class:  Thesis                               ;;
;;     Date:  14 November 93                       ;;
;;  File name: entity.clp                          ;;
;;                                                 ;;
;;  Purpose:  This provides the entity definitions ;;
;;  for the reasoning capbility for entities       ;;
;;  performing a  bomb target and a simple route   ;;
;;  planning mission.  It was written using the    ;;
;;  features of the CLIPS Object Oriented Language ;;
;;  (COOL), CLIPS version 5.0.                     ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  ASSUMPTIONS:                                   ;;
;;  1.  Only five icon defintions have been defined ;;
;;      for BATTLESIM so far.  The assumption is   ;;
;;      that the numbering of the icon definitions ;;
;;      will remain as currently used by BATTLESIM ;;
;;      and presented in Bergman's Thesis:         ;;
;;     1 - f18                                     ;;
;;     2 - mig1                                    ;;
;;     3 - missile                                 ;;
;;     4 - tank                                    ;;
;;     5 - truck                                   ;;
;;                                                 ;;
;;  2.  Changes to this numbering scheme will cause ;;
;;      adjustments to be made in the slot ''id''  ;;
;;      default values in the objects.             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Define some global variables to hold information    ;;
;;  needed by other functions throughout the reasoning  ;;
;;  process.                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;
;;  Super Class  ;;
;;;;;;;;;;;;;;;;;;
(defclass base-entity
   (is-a USER)
   (slot nomenclature)
   (slot type)
   (slot id)
   (slot speed)
   (slot fuel_capacity)
   (slot mobility)
   (slot min_path_width)
   (slot max_climbing_height)
   (slot max_ford_depth)
   (slot size)
   (slot mass)
   (slot sensors)
   (slot sensor_type)
   (slot sensor_range)
   (slot sensor_resolution)
   (slot armaments)
```

```
   (slot armament_description)
   (slot defensive_systems)
   (slot defensive_sys_descr)
   (slot loyalty)
   (slot fuel_status)
   (slot condition)
   (slot vulnerability)
   (slot x_velocity)
   (slot y_velocity)
   (slot z_velocity)
   (slot yaw_rate)
   (slot pitch_rate)
   (slot roll_rate)
   (slot experience)
   (slot threat_knowledge)
   (slot min_turn_rad)
   (slot avg_fuel_consump)
   (slot max_climb)
   (slot targets)
   (slot target_list)
)

;;;;;;;;;;;;;;;;;;;;;
;;  Tank Subclass  ;;
;;;;;;;;;;;;;;;;;;;;;
(defclass tank
   (is-a base-entity)
   (slot nomenclature          (default M1))
   (slot bumper_number)
   (slot type                  (default 4))
   (slot id                    (default 4))
   (slot speed                 (default 25))      ;; meters/sec = 15 mph
   (slot fuel_capacity         (default 500))    ;; 500 gallons
   (slot mobility              (default 10))
   (slot min_path_width        (default 30))
   (slot max_climbing_height   (default 1))
   (slot max_ford_depth        (default 2))
   (slot size                  (default 1))      ;;  10 meters
   (slot mass                  (default 60000))  ;;  kilograms
   (slot sensors               (default 1))
   (slot sensor_type           (default 1))
   (slot sensor_range          (default 5850))
   (slot sensor_resolution     (default 1))
   (slot armaments             (default 0))
   (slot armament_description (default ""))
   (slot defensive_systems     (default 0))
   (slot defensive_sys_descr   (default ""))
   (slot loyalty               (default 1))
   (slot fuel_status           (default 1))
   (slot condition             (default 1))
   (slot vulnerability         (default 1))
   (slot x_velocity            (default 25))      ;; meters/sec = 15 mph
   (slot y_velocity            (default 0))
   (slot z_velocity            (default 0))
   (slot yaw_rate              (default 0))
   (slot pitch_rate            (default 0))
   (slot roll_rate             (default 0))
   (slot experience            (default 1))
   (slot threat_knowledge      (default 1))
   (slot min_turn_rad          (default 1))
```

```
   (slot avg_fuel_consump     (default 1))
   (slot max_climb            (default 1))
   (slot targets              (default 0))
   (slot target_list          (default ""))

)
;;;;;;;;;;;;;;;;;;;;
;;  Truck Subclass  ;;
;;;;;;;;;;;;;;;;;;;;
(defclass truck
   (is-a base-entity)
   (slot nomenclature         (default HMMWV))
   (slot bumper_number)
   (slot type                 (default 5))
   (slot id                   (default 5))
   (slot speed                (default 60))      ;; meters/sec = 35 mph
   (slot fuel_capacity        (default 30))      ;; 30 gallons
   (slot mobility             (default 5850))
   (slot min_path_width       (default 20))
   (slot max_climbing_height  (default .5))
   (slot max_ford_depth       (default 1))
   (slot size                 (default 1))       ;; 10 meters
   (slot mass                 (default 1000))    ;; kilograms
   (slot sensors              (default 1))
   (slot sensor_type          (default 1))
   (slot sensor_range         (default 5850))
   (slot sensor_resolution    (default 1))
   (slot armaments            (default 0))
   (slot armament_description (default ""))
   (slot defensive_systems    (default 0))
   (slot defensive_sys_descr  (default ""))
   (slot loyalty              (default 1))
   (slot fuel_status          (default 1))
   (slot condition            (default 1))
   (slot vulnerability        (default 1))
   (slot x_velocity           (default 18))
   (slot y_velocity           (default 0))
   (slot z_velocity           (default 0))
   (slot yaw_rate             (default 0))
   (slot pitch_rate           (default 0))
   (slot roll_rate            (default 1))
   (slot experience           (default 1))
   (slot threat_knowledge     (default 1))
   (slot min_turn_rad         (default 1))
   (slot avg_fuel_consump     (default 1))
   (slot max_climb            (default 1))
   (slot targets              (default 0))
   (slot target_list          (default ""))

)

;;;;;;;;;;;;;;;;;;;;
;;  Plane Subclass  ;;
;;;;;;;;;;;;;;;;;;;;
(defclass plane
   (is-a base-entity)
   (slot nomenclature         (default F18))
   (slot bumper_number)
   (slot type                 (default 1))
   (slot id                   (default 1))
```

```
    (slot speed                 (default 1000))       ;;meters/sec = 500 mph
    (slot fuel_capacity         (default 1000))      ;; 1000 lbs
    (slot mobility              (default 20))
    (slot min_path_width        (default 30))
    (slot max_climbing_height   (default 12200))     ;;meters = 40k ft
    (slot max_ford_depth        (default 2))
    (slot size                  (default 1))        ;; 10 meters
    (slot mass                  (default 2000))     ;; kilograms
    (slot sensors               (default 1))
    (slot sensor_type           (default 1))
    (slot sensor_range          (default 5850))
    (slot sensor_resolution     (default 1))
    (slot armaments             (default 0))
    (slot armament_description  (default ""))
    (slot defensive_systems     (default 0))
    (slot defensive_sys_descr   (default ""))
    (slot loyalty               (default 1))
    (slot fuel_status           (default 1))
    (slot condition             (default 1))
    (slot vulnerability         (default 1))
    (slot x_velocity            (default 1))
    (slot y_velocity            (default 1))
    (slot z_velocity            (default 1))
    (slot yaw_rate              (default 1))
    (slot pitch_rate            (default 1))
    (slot roll_rate             (default 1))
    (slot experience            (default 1))
    (slot threat_knowledge      (default 1))
    (slot min_turn_rad          (default 1))
    (slot avg_fuel_consump      (default 1))
    (slot max_climb             (default 1))
    (slot targets               (default 0))
    (slot target_list           (default ""))
)
```

Appendix H. Sample Mission File

```
1 A22 tank M1A1 route-recon 221200zAugust93 00070007 00080023
2 B23 plane MIG1 route-recon 221200zAugust93 00080023 00070007
3 T123 plane F18 bomb-target 221400zAugust93 08990205 08680230
```

Appendix I. Generated BATTLESIM Scenario File

```
* version number
V4.0
* terrain data filename
terrain.10
* terrain min coordinates (x, y, z)
0.1 0.1 0.1
* terrain max coordinates (x, y, z)
117000.0 118000.0 1000.0
* number of sectors (must be < 64)
8
* sector min/max boundaries (x, y, z values in order from 1st to last sectors)
0.1 59000. 0.1 29250. 118000. 1000.
29250. 59000. 0.1 58500. 118000. 1000.
58500. 59000. 0.1 87750. 118000. 1000.
87750. 59000. 0.1 117000. 118000. 1000.
0.1 0.1 0.1 29250. 59000. 1000.
29250. 0.1 0.1 58500. 59000. 1000.
58500. 0.1 0.1 87750. 59000. 1000.
87750. 0.1 0.1 117000. 59000. 1000.
* number of icon records
5
1 f18
2 mig1
3 missile
4 tank
5 truck
* object type thru max climb
4 1 1 0 1 1 1 25 0 0 0 0 0 1 1 1 25 1 1
* number of route points
17
* route coordinates x, y, z (start to finish order)
7.0, 7.0, 100.0
6.0, 8.0, 100.0
6.0, 9.0, 100.0
6.0, 10.0, 100.0
6.0, 11.0, 100.0
6.0, 12.0, 100.0
6.0, 13.0, 100.0
6.0, 14.0, 100.0
6.0, 15.0, 100.0
6.0, 16.0, 100.0
7.0, 17.0, 100.0
7.0, 18.0, 100.0
7.0, 19.0, 100.0
```

```
7.0, 20.0, 100.0
8.0, 21.0, 100.0
8.0, 22.0, 100.0
8.0, 23.0, 100.0

* number of sensors
1
1 5850 1
* number of armaments
0
* armaments description (if above > 0)

* number of targets
0
* target descriptions (if above > 0)

* number of defensive systems
0
* defensive systems descriptions (if above > 0)

* END OF OBJECT
* object type thru max climb
1 2 1 0 1 1 1 1 1 1 1 1 1 1 1 1000 1 1
* number of route points
17
* route coordinates x, y, z (start to finish order)
8.0, 23.0, 100.0
8.0, 22.0, 100.0
7.0, 21.0, 100.0
7.0, 20.0, 100.0
7.0, 19.0, 100.0
7.0, 18.0, 100.0
6.0, 17.0, 100.0
6.0, 16.0, 100.0
6.0, 15.0, 100.0
6.0, 14.0, 100.0
6.0, 13.0, 100.0
6.0, 12.0, 100.0
6.0, 11.0, 100.0
6.0, 10.0, 100.0
6.0, 9.0, 100.0
7.0, 8.0, 100.0
7.0, 7.0, 100.0

* number of sensors
```

```
1
1 5850 1
* number of armaments
0
* armaments description (if above > 0)

* number of targets
0
* target descriptions (if above > 0)

* number of defensive systems
0
* defensive systems descriptions (if above > 0)

* END OF OBJECT
* object type thru max climb
1 3 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1000 1 1
* number of route points
63
* route coordinates x, y, z (start to finish order)
899.0, 205.0, 100.0
898.0, 206.0, 100.0
897.0, 207.0, 100.0
896.0, 207.0, 100.0
895.0, 208.0, 100.0
894.0, 209.0, 100.0
893.0, 210.0, 100.0
892.0, 211.0, 100.0
891.0, 211.0, 100.0
890.0, 212.0, 100.0
889.0, 213.0, 100.0
888.0, 214.0, 100.0
887.0, 215.0, 100.0
886.0, 215.0, 100.0
885.0, 216.0, 100.0
884.0, 217.0, 100.0
883.0, 218.0, 100.0
882.0, 219.0, 100.0
881.0, 220.0, 100.0
880.0, 220.0, 100.0
879.0, 221.0, 100.0
878.0, 222.0, 100.0
877.0, 223.0, 100.0
876.0, 224.0, 100.0
875.0, 224.0, 100.0
```

```
874.0, 225.0, 100.0
873.0, 226.0, 100.0
872.0, 227.0, 100.0
871.0, 228.0, 100.0
870.0, 228.0, 100.0
869.0, 229.0, 100.0
868.0, 230.0, 100.0
869.0, 229.0, 100.0
870.0, 228.0, 100.0
871.0, 228.0, 100.0
872.0, 227.0, 100.0
873.0, 226.0, 100.0
874.0, 225.0, 100.0
875.0, 224.0, 100.0
876.0, 224.0, 100.0
877.0, 223.0, 100.0
878.0, 222.0, 100.0
879.0, 221.0, 100.0
880.0, 220.0, 100.0
881.0, 220.0, 100.0
882.0, 219.0, 100.0
883.0, 218.0, 100.0
884.0, 217.0, 100.0
885.0, 216.0, 100.0
886.0, 215.0, 100.0
887.0, 215.0, 100.0
888.0, 214.0, 100.0
889.0, 213.0, 100.0
890.0, 212.0, 100.0
891.0, 211.0, 100.0
892.0, 211.0, 100.0
893.0, 210.0, 100.0
894.0, 209.0, 100.0
895.0, 208.0, 100.0
896.0, 207.0, 100.0
897.0, 207.0, 100.0
898.0, 206.0, 100.0
899.0, 205.0, 100.0

* number of sensors
1
1 5850 1
* number of armaments
0
* armaments description (if above > 0)
```

```
* number of targets
0
* target descriptions (if above > 0)

* number of defensive systems
0
* defensive systems descriptions (if above > 0)

* END OF OBJECT
```

```
/*****************************/
/*          getgrid.c        */
/*                           */
/*  Date:  25 May 93         */
/*  The filelength check is  */
/*  nonportable from MS-DOS  */
/*  to UNIX.                 */
/*  Revisions:    23 May 93  */
/*  Changed record length to */
/*  20 bytes:                */
/*  X coordinate - 4 bytes   */
/*  Y coordinate - 4 bytes   */
/*  Z coordinate - 4 bytes   */
/*  Vegetation   - 3 bytes   */
/*  Mobility     - 3 bytes   */
/*  spare        - 2 bytes   */
/*               ----------  */
/*          Total 20 bytes   */
/*                           */
/* Changed offset pointer to */
/* long integer              */
/*           17 June 93      */
/*   Took out two uneeded    */
/*  declarations.            */
/*****************************/
/*************************************************************/
/*  This function opens the terrain file and reads in        */
/*  the information for a particular grid square. The        */
/*  terrain file is stored in contiguous record to           */
/*  allow direct access to any record.                       */
/*************************************************************/
#include <stdio.h>
#include <string.h>
#include "clips.h"
#include "constant.h"
#include "thesis.h"
char *getgrid()
{
FILE *terrain_file_ptr;
char *returnValue;
char gridrecord[21];
long offset;
long grid;
grid = RtnLong(1);
if ((terrain_file_ptr = fopen("terrain.fil", "r")) == NULL) {
 printf("terrain.fil not found\n");
}
else {
offset = grid * 20;
fseek(terrain_file_ptr, offset, 0);
fgets(gridrecord, 21, terrain_file_ptr);
returnValue = AddSymbol(gridrecord);
}
fclose(terrain_file_ptr);
return (returnValue);
}
```

Appendix K. Users Guide

K.1 Introduction

The Automated Scenario Generation System runs under CLIPS 5.0 and needs the following files to execute properly:

- CLIPS version 5.0 or higher
- scenhead.fil - holds the BATTLESIM version number and sectoring information.
- terrain.fil - terrain database currently used by the Automated Scenario Generation System.
- mission.fil - specifies objects, key points and missions for the scenario.
- scenario.fil - this file needs to be deleted prior to every run or new results will be appended to previous runs.

K.2 Execution

First, start CLIPS5. At the CLIPS prompt type **(load asgs.clp)**. This will cause CLIPS to load in the asgs.clp program. After CLIPS finishes, type **(asgs)**. This will cause the entity descriptions, mission definitions and driver programs to be loaded into CLIPS. After this is complete, type **(execute-mission)**.

The Automated Scenario Generation System will then begin reading the mission file, instantiating entities and assigning them missions for planning. The entities will conduct their missions and return the results to the driver program to write to the scenario file. This will continue until all missions in the mission file are completed. When all missions are complete the user will be returned to the **CLIPS>** prompt.

To exit type **(exit)** and the user will exit CLIPS.

Appendix L. Text Processor Prototype

```
#     Natural Language Processor
#
#    Author:  Mark W. Pfefferman
#      Date:  2 October 93
#  Filename:  nlp.awk
#   Purpose:  This routine is a simple prototype
#             text processer.  The routine reads
#             in the file oporder.fil, processes
#             it and creates a mission file.

BEGIN {
        j = 0
        RS = "."
}
/Situation/
{
        printf("Situation paragraph found\n %s", $0)
        for (i=1; i<= NF;i++) {
              if ($i == "Armored" || $i == "armored")
              {
              unit = "tank"
              type = "M1A1"
              }
              if ($i == "Wing")
              {
              unit = "plane"
              type = "F18"
              }
        }
}
/Mission/ {print "Mission paragraph found " NR}
/Execution/
{
        print "Execution paragraph found " NR
        for (i=1; i <= NF; i++) {
              if ($i == "conduct" )
              {
                  print "what " $(i+2)
                # who
                  j++
                  if (j==1)
   printf("%d %s %s %s", j, $(i-7), type, unit) >> "mission.fil";
   else
   printf("\n%d %s %s %s", j, $(i-7), type, unit) >> "mission.fil"
                # what
                  printf(" %s", $(i+2)) >> "mission.fil"
              }
              if ($i == "vicinity" && $(i-2) != "Checkpoint")
              {
                  print "where " $(i+1)
                # start
                  printf(" %s", $(i+1)) >> "mission.fil"
              }
              if ($i == "Checkpoint" || $i == "checkpoint")
              {
                  print "through " $(i+3)
                  printf(" %s", $(i+3)) >> "mission.fil"
              }
```

```
                if ($i == "commencing")
                {
                    print "when"
                    printf(" %s", $(i+2)) >> "mission.fil"
                }
        }
}
/Service/ {print "Service Support paragraph found " }
/Command / {print "Command and Signal Paragraph found " }
END {}
```

Appendix M. Sample Operation Order

I. Situation.

A. Enemy Forces. Elements of the Iraqi 5th Republican Guards Division have taken up positions near the town of An Nasiriyah. The 10th Special Forces Brigade has been located near the Saudi Arabian town of Rafah.

B. Friendly Forces. The 1st Armored Division will have the 1-101st Aviation Battalion (AH-64) and the 197th Separate Armor Brigade in direct support during this operation. Attachment is effective 010000z_August_93 and will be terminated at the end of the operation at the discretion of the 1st Cavalry Division Commander.

II. Mission. A22 will perform a route-recon mission NLT 041200z_August_93 from its present location on Hill 744 to Hill 900. T123 of the 85th Bomber Wing
will perform a bomb-target mission NLT 040800z_August_93 its present location at the King Fahd airfield to Hill 900 to neutralize any enemy forces located there.

III. Execution. A22 of the 1st Armored Division will conduct a route-recon mission commencing at 041200z_August_93 from its present location on Hill 744 in Assembly Area Blue, vicinity AAAAAAAA to Hill 900, vicinity BBBBBBBB through Checkpoint Detroit vicinity CCCCCCCC.

T123 of the 85th Bomber Wing will conduct a bomb-target mission commencing at 040800z_August_93 from its present location at the King Fahd airfield vicinity 11111111 to Hill 900 vicinity 22222222 through Checkpoint Anaheim vicinity 33333333 on the route out and through Checkpoint Baltimore vicinity 44444444 on the return route.

B23 of the 1st Armored Division will conduct a route-recon mission commmencing at 040800z_August_93 from its present location vicinity XXXXXXXX to determine the trafficability of Highway 1 to Baghdad, vicinity YYYYYYYY.

IV. Service Support.

A. All classes of supply are currently available in the Division Support Area at Forward Operating Base (FOB) Bastogne. No significant shortages of any class is expected for the duration of the operation.

B. See Annex D - Service Support for more specific guidance.

V. Command and Signal.

A. Command. Command Post Locations will be published separately.

B. Signal.

1. Call signs and frequencies will change over at 1200 hours local, 03 August 93 as normal.

2. Call signs and frequencies will remain frozen on Day 4 through Phase III of the operation.

3. See Annex K - Signal for radio, satellite and telephone network information.

Appendix N. Resulting Mission File

```
1 A22 M1A1 tank route-recon 041200z_August_93 AAAAAAAA BBBBBBBB CCCCCCCC
2 T123 F18 plane bomb-target 040800z_August_93 11111111 22222222 33333333 44444444
3 B23 M1A1 tank route-recon XXXXXXXX YYYYYYYY
```

Bibliography

1. Asano, Takao and others. "Shortest Path Between Two Simple Polygons," *Information Processing Letters*, *24*:285–288 (March 1987).
2. Bergman, Captain Kenneth C. *Spatial Partitioning of a Battlefield Parallel Discrete-Event Simulation*. MS thesis, Air Force Institute of Technology, 1992.
3. Cunningham, C. T. "Control of Movement in an Arbitrary Polygonal Terrain." *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation*. 307–315. March 1993.
4. Department of the Army. *FM 7-10 The Infantry Rifle Company (Infantry, Airborne, Air Assault, Ranger*, March 1982.
5. Department of the Army. *FM 7-7 The Mechanized Infantry Platoon and Squad*, March 1985.
6. Department of the Army. US Army Training and Doctrine Command (TRADOC) Analysis Command, Fort Leavenworth, KS. *Project Eagle Overview Briefing Packet*.
7. DeRouchey, William. *A Remote Visual Interface Tool for Simulation Control and Display*. MS thesis, Air Force Institute of Technology, 1990.
8. Downes-Martin, Stephen. *Open and Extensible Architecture for Computer Generated Forces*. Loral Systems Company, Orlando, Florida, February 1992.
9. Foley, James D. and others. *Computer Graphics: Principles and Practice*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1990.
10. Hartman, James K. *Introduction to Models of Combat Lecture Notes*. Naval Post Graduate School, Monterey CA, 1985.
11. Hartrum, Dr Thomas C. *Lecture on BATTLESIM Model, Parallel Simulations Working Group*. School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, November 1992.
12. Hoffman, LTC Camillus W.D. *CASTFOREM (Combined Arms and Support Task Force Evaluation Model Update: Scenario Writers Guide*. Department of the Army, US Army TRADOC Analysis Command, White Sands Missile Range, NM, 1992.
13. Jackson, James O. "Thanks, but No Tanks," *Time*, 21–22 (February 5 1990).
14. Joshi, Aravind K. "Natural Language Processing," *Science*, 1242–1248 (September 1991).
15. Kilpatrick, Captain Freeman A. *An Investigation of Discovery-Based Learning in the Route Planning Domain*. MS thesis, Air Force Institute of Technology, 1992.
16. Negrelli, Major Edward P. *Lecture Notes In High Resolution Combat Modelling OPER 775*. School of Operational Sciences, Air Force Institute of Technology, Wright-Patterson AFB OH, September 1992.
17. Powell, Dennis R. and LTC John L. Hutchinson. "Eagle II: A Prototype for Multi-Resolution Combat Modeling." *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation*. 221–230. March 1993.

18. "RASPUTIN – Systems Requirement Specification," March 1991.

19. Rich, Elaine and Kevin Knight. *Artificial Intelligence*. New York: McGraw-Hill, Inc., 1991.

20. Ross, Major Ron S. *Planning Minimum-Energy Paths in an Off-Road Environment with Anisotropic Traversal Costs and Motion Constraints*. PhD dissertation, Naval Postgraduate School, 1989.

21. Rumbaugh, James and others. *Object-Oriented Modeling and Design*. Englewood Cliffs: Prentice-Hall, Inc., 1991.

22. Salisbury, Marnie and Hans Tallis. "Automated Planning and Replanning for Battlefield Simulation." *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation*. 243-254. March 1993.

23. Software Technology Branch Lyndon B. Johnson Space Center. *CLIPS Basic Programming Guide*
CLIPS Version 5.1, September 1, 1991.

24. Stone, Michael P. W. "The Challenge: To Reshape, Remain Army Excellence," *AUSA 1991 ARMY Green Book*, 12-20 (1991).

25. Sun Tzu. *The Art of War*. New York: Delacote Press, 1983.

26. TeKnowledge Inc, "TeKnowledge's M4 User's Guide," September 1, 1991.

Vita

Captain Mark W. Pfefferman was born February 19, 1962, in Kentucky. After graduating from Campbell County High School in 1980, he enrolled in Western Kentucky University. He graduated in May 1984 with a Bachelor of Science Degree in Computer Science and was commissioned as a second lieutenant in the Signal Corps, United States Army. He has served in Germany and a variety of state side posts. He served as a company commander during Desert Shield and Desert Storm while assigned to the 101st Airborne Division (Air Assault). He was the Division Radio Officer for the 101st prior to his assignment to the Air Force Institute of Technology.

Permanent address: 119 Creekstone Court
Cold Springs, Kentucky 41076

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1993	Master's Thesis

4. TITLE AND SUBTITLE
A PROTOTYPE ARCHITECTURE FOR AN AUTOMATED SCENARIO GENERATION SYSTEM FOR COMBAT SIMULATIONS

5. FUNDING NUMBERS

6. AUTHOR(S)
Mark W. Pfefferman, Captain, USAR

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/GCS93D-18

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This research effort examines the problem of automating the scenario generation process and presents a prototype architecture for an automated scenario generation system. This architecture is designed using an object-oriented approach which leads to a modular and modifiable design. The architecture provides a mechanism for automatically generating scenario files from a textual US Army operation order. This translation process occurs in two phases. First, the text operation order is translated into an intermediate format called the mission file. In the second phase, the system reads the mission file, instantiates intelligent entities, and assigns missions to those entities. The intelligent entities emulate the subordinate leaders in planning missions. The thesis effort assumes that sufficiently sophisticated natural language processing algorithms can translate the operation order into the mission file. A prototype scenario generation system was implemented using NASA's CLIPS expert system development tool. The prototype system successfully translates mission files into scenario files for the BATTLESIM combat simulation model developed at the Air Force Institute of Technology.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Automated Scenario Generation, CLIPS, mission planning, combat simulations, war-games, COOL, artificial intelligence, expert systems	**112**
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	**Unclassified**	**Unclassified**	**UL**